P Systems as Formal Models for Distributed Algorithms

Huiling Wu

under the supervision of

Dr. Radu Nicolescu and Prof. Tudor Bălănescu

A thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science,
The University of Auckland, 2013

Abstract

A P system is a parallel and distributed computational model inspired by the structure and interactions of living cells [55].

Chapter 1 first relates mainstream concepts of distributed computing to distributed features proposed for P systems. Chapter 2 introduces *xP systems*, which are extended versions of simple P systems proposed in our joint work [51, 3, 52, 22, 50].

This thesis investigates the adequacy of xP systems for modelling fundamental synchronous and asynchronous distributed algorithms and aims to construct *xP specifications*, i.e. directly executable *formal* specifications of algorithms in xP systems, which: (1) achieve the same runtime complexities as corresponding distributed algorithms and (2) are comparable in program size with high-level pseudocodes of corresponding distributed algorithms.

Chapter 3 presents xP specifications of several fundamental traversal algorithms: Echo, distributed depth-first search (DFS), distributed breadth-first search (BFS) and neighbour discovery algorithms, based on our joint work [3]. As new contributions in this thesis, we* address the common problem in distributed algorithms, termination detection problem, and provide xP specifications of several well-known termination detection algorithms and their applications.

Chapter 4 discusses a series of distributed synchronous DFS and BFS-based edge- and node-disjoint paths algorithms. Consider a digraph with n nodes and m arcs, where f is the maximum number of disjoint paths and d is the outdegree of the source node. Dinneen et al.'s [18] algorithms based on the classical DFS run in $O(mf)$; using Cidon's DFS and Theorem 4.5, our improved algorithms [52] run in $O(nf)$; using a different idea, our two other DFS-based algorithms [22] run in $O(nd)$ and $O(nf)$. The first BFS-based P system solutions in our joint work [51, 52] run in $O(nf)$; an improved version as my own work [65] also runs in $O(nf)$.

Following my own work [64], Chapter 5 presents a solution for one of the most challenging distributed computing problems: minimum spanning tree (MST) problem. We discuss the SynchGHS algorithm [42] and our synchronisation barriers. Given a weighted graph with n nodes, our xP solution runs in $O(n \log n)$ and it is the first MST solution in P systems.

*Following Knuth et al.'s advice [38], I use "we" rather than "I" in this thesis for inviting the reader to be part of this "journey". However, I confirm that, Huiling Wu, was the sole author of this thesis.

Acknowledgments

First and foremost, I would like to thank my supervisor, Dr. Radu Nicolescu, for his guidance and constant advice during my study. Over the last four years, I have benefited immensely from his wisdom and advice. Without his kind consideration of my situation change, I would have not been able to complete my thesis.

I am grateful to my co-supervisor, Prof. Tudor Bălănescu, for his advice and reviews of my research papers.

My greatest gratitude goes to my family, for their understanding and unconditional support. I am thankful to my parents, for their encouragement and the care of my daughter during these difficult years. I am thankful to my husband, for his patience and the financial support of the whole family. I am thankful to my four-year-old daughter, for her understanding during my research and thesis writing.

I would like to thank the University of Auckland, for the scholarships and grant during the period of my study.

I wish to thank the two examiners' detailed comments and feedback that helped me improve my thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Distributed Systems

A distributed system is an interconnected collection of autonomous *computing elements* (often logically called *processes*), such as computers, processes and processors. Processes in a distributed system communicate by *shared memory* or *message passing*.

This thesis restrictively considers distributed message-based (no shared memory) systems where processes communicate by messaging passing. Thus, a distributed system consists of a collection of processes and a *communication subsystem*. Each process performs a collection of discrete *events*, each event being an atomic change. To interact with the communication subsystem, a process has *internal events*, which perform local computations, *receive events*, which receive messages from channels, and *send events*, which queue messages to appropriate channels [62].

Network: The *network* is a digraph, where an arc exists if and only if a communication *channel* between two processes exists. The digraph can be a directed acyclic graph (dag), a tree, etc.

Channel: Channels between processes can be *simplex* (unidirectional) or *duplex* (bidirectional). A channel is *reliable* if every message sent is received exactly once; a channel is *FIFO* if it keeps the order of the messages sent through it.

Computation: Each process repeatedly performs computation of the following three substeps, each consisting of one or more events:

- *Receive* substep: executes receive events, if any;
- *Process* substep: executes internal events, if any; and
- *Send* substep: executes send events, if any.

There are various conventions of the execution order of these three substeps, e.g., Lynch's Send-Receive-Process (SRP) [42] and Tel's Process-Send-Receive (PSR) [62]. This thesis uses the common order [29], Receive-Process-Send (RPS), which suits the common time complexity measure, as later discussed.

For all distributed algorithms in this thesis, RPS substeps are performed as an *atomic* step, which occurs instantaneously.

Synchrony and asynchrony: Messaging can be *synchronous* or *asynchronous*. A synchronous system has a global clock and each process performs a step consisting of *Receive*, *Process* and *Send* substeps at each clock tick. However, an asynchronous system does not have such a global clock and there can be an arbitrary transmission delay between the sending and receipt of a message.

Process activity status: At any time during the computation of a distributed algorithm, a process is either

- *active*, if an internal or send event is applicable (i.e. *Process* or *Send* substep is applicable); or
- *passive*, if no internal or send event is applicable: only receive events are applicable (i.e. only *Receive* substep is applicable) [62].

Initiator: A process is an *initiator* if it is active when the computation starts. A distributed system can have one or more initiators.

Activation assumptions: To simplify the descriptions of distributed algorithms in this thesis, the following *activation assumptions* are made:

- internal events can be only activated or disactived by receive events; and
- send events can be only activated by internal events.

As a consequence,

- a message can only be sent by an active process;
- a passive process can only become active when a message is received;
- an active process can only become passive after performing an internal event or a send event.

Figure 1.1 shows the chart of process activity status transition under the above assumptions.

Termination: A distributed algorithm *terminates* when all processes are passive and all channels are empty. Termination can be

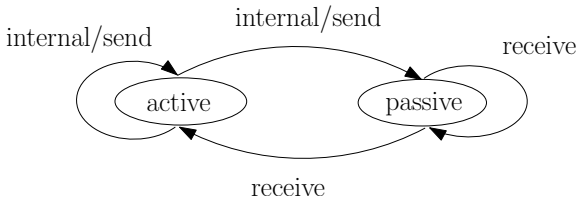


Figure 1.1: Process activity status chart.

- *implicit*: no process knows termination;
- *explicit*: all processes know termination;
- *partial-explicit*: for the algorithm having one single initiator, the initiator knows but not necessarily all processes know termination.

Termination detection and *announcement* aim to convert an implicitly terminating algorithm to an explicit terminating algorithm (later discussed in Section 3.6).

Message complexity: Message complexity measures the total number of messages used by the system.

Runtime complexity: Various definitions can be used to measure the time complexity of a distributed algorithm.

Definition 1.1 (Common time complexity). The complexity of a distributed algorithm is the maximum time taken by any computation of the algorithm, under the following assumptions [62].

1. A process can execute any finite number of events in zero time.
2. The transmission delay, t, between the sending and receipt of a message is a non-negative real number, which is *at most* one time unit, i.e. $t \in [0, 1]$.

Note that execution order RPS seems the most suitable for the common time complexity measure: there is no delay between Receive, Process and Send so the execution of events takes zero time. In contrast, in SRP and PSR, there can be arbitrary delay between Send and Receive, so the execution of events takes more than zero time.

Definition 1.2 (One-time complexity). The one-time complexity of a distributed algorithm is the maximum time taken by any computation of the algorithm under the following assumptions [62].

1. A process can execute any finite number of events in zero time.
2. The transmission delay, t, between the sending and receipt of a message is *exactly* one time unit, i.e. $t = 1$.

One-time complexity only considers *synchronous* computations.

The set of all computations under assumptions of one-time complexity is a subset of the set of all computations under assumptions of common time complexity. Thus, one-time complexity is less or equal than common time complexity [62].

Definition 1.3 (Chain-time complexity). A message chain is a sequence $m_1, m_2, \ldots, m_k$ of messages, such that for each $i, 0 \leq i < k$, the receipt of m_i causally precedes the sending of m_{i+1} (see Tel's book [62] for more details on causal order). The chain-time complexity of a distributed algorithm is the length of the longest message chain in any computation of the algorithm [62].

Chain-time complexity and common time complexity consider all possible computations of an algorithm, even though some are unlikely to occur, as later seen in Figure 1.3. For a system where transmission delays have an upperbound, common time complexity is a suitable measure; for a system where only few transmission delays are very large, chain-time complexity is recommended [62].

Example 1.1 illustrates these concepts with the same Echo algorithm in two different distributed runtime scenarios: (1) synchronously (see Figure 1.2) and (2) asynchronously (see Figure 1.3) [3].

Example 1.1. The Echo algorithm [62] starts from a source, which broadcasts *visit tokens*. These visit tokens transitively reach all processes and, at the end, are reflected back to the source. The *forward* phase establishes a virtual spanning tree and the *return* phase is supposed to follow up its branches. The algorithm terminates when the source receives all expected visit tokens.

Consider a graph, G, where n is the number of nodes and diam is the diameter, which is the greatest distance between any pair of nodes [16]. Scenario 1 in Figure 1.2 assumes that all messages arrive in *exactly* one time unit, $t = 1$, i.e. in the *synchronous* mode. The forward phase takes $\text{diam} + 1$ time units and the return phase takes diam time units. Thus, the runtime of this computation measured under the assumptions of Definition 1.1, 1.2 and 1.3 is the same, i.e. $2\,\text{diam} + 1$.

Scenario 2 in Figure 1.3 assumes that some messages travel faster than others in the *asynchronous* mode. Under the assumptions of Definition 1.1, $t = \epsilon$, where $0 < \epsilon \ll 1$, the forward and return phases take very different times, diam and $n - 1$ time units respectively; thus the runtime is $\text{diam} + n - 1$. Under the assumptions of Definition 1.3, scenario 2 shows one of the longest message chains, which follows the path 1.2.3.4.1.4.3.2.1. The forward and return phases take the same time, n time units; thus the runtime is $2n$.

Specification: A *pseudocode* is an *informal* specification of an algorithm, which is *non-executable*. It contains essential descriptions of an algorithm and often omits many implementation details.

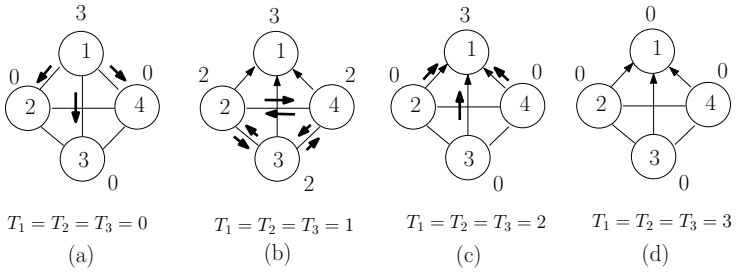


Figure 1.2: The Echo algorithm in the synchronous mode, when all messages are propagated with the same delay ($t = 1$). Edges with arrows: virtual spanning tree child-parent arcs; thick arrows near edges: visit tokens; the number near each process: the number of expected tokens. T_1, T_2 and T_3 are the runtime according to Definitions 1.1, 1.2 and 1.3, respectively.

In this thesis, we consider the program size of the pseudocode of an algorithm as the number of high-level pseudocode operations, which typically corresponds to the number of lines, in a well-structured and laid out pseudocode.

To improve the readability and understanding of *synchronous* distributed algorithms, Chapters 4 and 5 present *sequentialised* pseudocodes and *structural parallel* pseudocodes. Structural parallel pseudocodes look like sequentialised pseudocodes but enhanced with a few parallel structures:

(a) `fork` launches a *parallel* task running synchronously at the *same speed* with the task that starts it. Example 1.2 (a) shows that F starts task F_1, which runs in parallel with F's task (denoted by ...) synchronously at the same speed; and F does not wait for F_1 to finish.

(b) `parallel foreach` launches parallel tasks and use implicit joins to wait until all parallel tasks finish. Example 1.2 (b) shows that F starts one task F_i for each A_i in set S; all tasks F_i's run in parallel synchronously at the same speed and F waits until all tasks F_i's finish.

(c) `parallel fork` launches parallel tasks and use implicit joins to wait until all parallel tasks finish. Example 1.2 (c) shows that F starts tasks F_1, F_2 and F_3, all of which run in parallel synchronously at the same speed, and waits until tasks F_1, F_2 and F_3 finish.

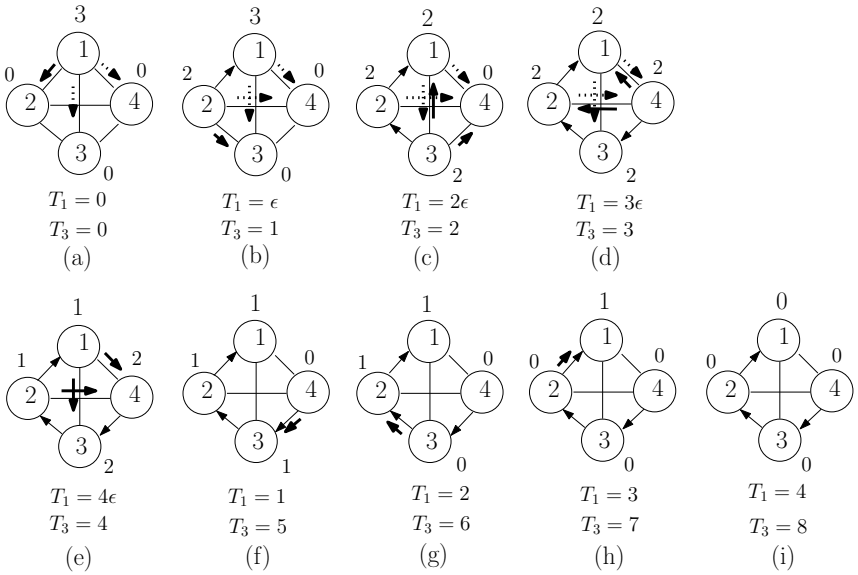


Figure 1.3: The Echo algorithm in one possible asynchronous run. Dotted thick arrows near edges: visit tokens still in transit; edges with arrows: virtual spanning tree child-parent arcs; thick arrows near edges: visit tokens; the number near each process: the number of expected tokens. T_1 and T_3 are the runtime according to Definitions 1.1 and 1.3, respectively.

Example 1.2.

```
F()
  fork F1()
  ...
```

(a)

```
F()
  parallel foreach Ai ∈ S
    Fi(Ai)
  endfor
```

(b)

```
F()
  parallel fork
    F1()
    F2()
    F3()
  endfork
```

(c)

1.2 P Systems

Membrane computing, introduced by Păun, is a bio-inspired research domain of computer science, aiming to abstract the ideas and computational models from the structure and interactions of living cells [55]. These computational models are called *P systems*. Previous extensions include Dinneen et al.'s hyperdag P systems [49] and simple P modules [17, 19] and our joint work of simple P systems [51, 3, 52, 22].

A P system is a parallel and distributed computational model consisting of a collection of autonomous computing elements, called *cells*. In this thesis, for consistency with the P system terminologies, processes in distributed systems and nodes in graph

theory are also called cells.

A P system consists of membranes arranged in a structure, e.g., a rooted tree in cell-like P systems [55], a digraph in neural P systems [56] and tissue P systems [44], or a directed acyclic graph in hyperdag P systems [49]. *Multisets of symbols* are placed in membranes. Cells evolve by means of applying *formal rewriting rules*, which transform its symbols and send messages to neighbours. Thus, the essential ingredients of a P system include its *membrane structure*, *symbols* and *rules*.

- **Membrane structure:** The underlying membrane structure is a *network*, which can be

  1. a digraph (neural P systems [56] and tissue P systems [44]),
  2. a more specialised version, such as a directed acyclic graph (hyperdag P systems [49]), or
  3. a rooted tree (cell-like P systems [55], which is one of the most studied cases).

  Channels can be simplex or duplex. Typically, it is assumed that channels are reliable and messages sent over the same channel arrive in strict *queue* order (FIFO)—but one can also consider arrival in *arbitrary* order (bag, instead of queue).

  The following assumptions are made in this thesis:

  - channels are duplex;
  - messaging is reliable (no modifications or loss) and messages arrive in FIFO order;
  - cells are reliable (no failures).

- **Symbols:** A cell contains a *multiset* of symbols (also called *content*). The multiset of symbols is represented as a string, e.g., a multiset of one symbol a and two symbols b is represented as string ab^2. Specifically, an empty multiset is represented by λ.

- **Rules:** Each cell transforms its content symbols and sends messages to its neighbours, by applying formal rules inspired by *formal rewriting systems* in a (potentially maximally) *parallel* manner. The *evolution* of all cells constitutes the evolution of a P system, exhibiting two levels of parallelisms:

  - for the same cell, applicable rules can be applied in parallel (where possible);
  - for the system, all cells evolve in parallel.

Each cell repeatedly performs a *step* consisting of the following three substeps, in Receive-Process-Send (RPS) order, where *Receive* substep is *automatically* done and *Process* and *Send* substeps are *combined* and performed by applying rules.

- *Receive* substep: receive symbols from channels, if any; this is automatically done and no rule is needed, i.e. a cell has to receive whatever is sent to it in P systems, in contrast with the *Receive* substep in distributed systems.
- Combined *Process* + *Send* substeps: apply all applicable rules that can be considered as combined internal+send events, which transform contents and send messages.

In our xP systems, RPS substeps are performed in one single step, which is an *atomic* step, as in distributed algorithms.

Synchrony and asynchrony: Traditional P systems are *synchronous*, where all cells' evolutions are controlled by a single global clock and each cell performs one step at each clock tick; while asynchronous P systems evolve without a global clock.

Cell activity status: At any time during the evolution of a P system, a cell is either

- *active*, if it has at least one applicable rule (combined *Process* + *Send* substeps are applicable); or
- *passive*, if it cannot apply any (more) rule (only *Receive* substep is applicable).

Activation assumptions:

P systems may not conform to activation assumptions (§ 1.1) in the usual distributed computing framework: a rule can be applicable at the start of next step without receiving a message. In this thesis, our xP systems are redefined to conform to usual activation assumptions, as later discussed in Section 2.3; thus the transitions of cell activity status can be shown as in Figure 1.4, adapted for P systems.

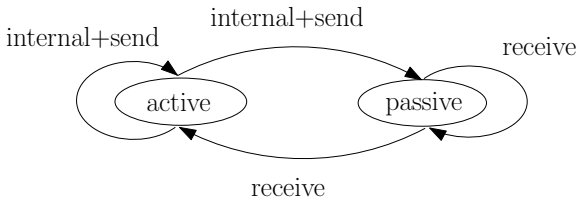


Figure 1.4: Cell activity status chart.

Initiator cell: A cell is an *initiator* cell, also called a *source* cell, if it is active when the P system starts to evolve.

Termination: A P system *terminates* when no rule is applicable and all channels are empty. Termination can be

- *implicit*: no cell knows termination;
- *explicit*: all cells know termination;
- *partial-explicit*: for the algorithm having one single source cell, the source cell knows but not necessarily all cells know termination.

Message complexity: The message complexity is the total number of messages transferred during the system evolution. A message can contain:

1. a single symbol (elementary or complex) by one rule to the target cell,
2. all symbols sent by one rule to the same target cell,
3. all symbols sent from a cell to the same target cell during a step (by all applicable rules).

This thesis uses measure 3, i.e. a message consists of all symbols sent from a cell to the same target cell during a step.

Runtime complexity: This thesis uses the measure proposed in our joint work [3], based on the common time complexity measure, where the runtime complexity of an asynchronous P system is defined as the maximum time over all possible evolutions, under the following assumptions:

1. for each cell, its rules application, once started, takes *zero* time (it occurs instantaneously);
2. for each message, its transmission delay, $t \in [0, 1]$, is *arbitrary*.

The runtime of a P system is also called *P time*. Specifically for a synchronous P system, $t = 1$, i.e. all transmission delays are one; thus its runtime complexity is the total number of *steps* of the computation.

Specification: A *P specification* is a *formal* specification of an algorithm in a P system and is directly *executable* in a P system simulator. A P specification consists a set of rules, i.e. a *ruleset*. A rule of a P specification is also called a *P rule*. A *P solution* is a P specification which solves a problem.

In this thesis, we consider the program size of a P system as the number of rules. Thus, a *P system size* is the number of rules in a P system.

In the context of our xP systems, a P specification is called an *xP specification*; a P rule is called an *xP rule*; a P solution is called an *xP solution*; a P system size is called an *xP system size*.

1.3 Motivation

Broadly speaking, typical P systems research falls into one of the following three areas [48, 57, 14]:

- Theory: computational completeness (universality), complexity classes (e.g., polynomial solutions for NP-hard problems) or relationships with other models (e.g., automata, grammar systems and formal languages). Many classes of P systems can achieve Turing computability. With exponential working space generated in polynomial time, P systems can achieve polynomial or even linear solutions to computational hard problems (e.g., NP-complete problems).
- Tools: designers, simulators and verifiers.
- Applications: many advanced applications are in the field of biology and biomedicine, and also in a large variety of other areas, such as approximate optimisation, computer graphics, robot control, cryptography and economics modelling.

Studies in P systems have achieved many interesting theoretical results. This thesis focuses on the practical applications of P systems, by modelling distributed algorithms.

Although theoretically possible, from our preliminary modelling experience, we find it difficult to model complex algorithms using the existing P systems framework.

Large program size: For complex distributed problems, traditional P systems can not achieve one single solution; instead, they provide a *uniform family* of solutions using variable size alphabets and rulesets. Using complex symbols (§ 2.3.3), we can achieve one single solution with a fixed-size alphabet and ruleset. Previous work on Byzantine problem [19] shows that the solution using complex symbols can reduce the solution size from $O((N)_L N)$ to $O(N)$, where N is the number of processes connected in a complete graph and L is the number of messaging rounds. For the SAT problem [47], all proposed traditional P system solutions require an exponential runtime number of cells and a polynomial number of symbols and rules, while P systems with complex symbols achieve a fixed-size alphabet and ruleset proportional to the number of variables of the formula [53].

Cell IDs: Traditional P systems do not use *cell IDs*, which are needed in distributed algorithms. Lynch [42] and Tel [62] highlight this need in the well-known distributed leader election problem and its impossibility result: it is *impossible* to deterministically elect a leader for a network of nodes, if the system is totally *symmetric*.

Step granularity: From our previous experience [22, 64], in traditional P systems, a step is too small to model all the work required to be done by a given complex

algorithm in a logical step. Experiments have also shown that it is practically impossible to maintain the expected runtime complexity within the traditional framework because of these limits. These limits can create serious issues, after a single step, if a cell that did not complete the processing of its already received message, it risks receiving more messages, which may create havoc. Typical distributed algorithms require a fair amount of work done in a single step, i.e. an *atomic* step, before succeeding messages arrive.

Other existing proposals for asynchronicity P systems: P systems with various asynchronous features have been investigated by recent research. Although theoretically interesting, they do not emulate asynchronous systems in the sense of distributed algorithms.

Some asynchronous P systems attempt to simulate asynchronous behaviour on top of traditional synchronous P systems. Simulation still runs in *integer* time units, where rule execution time takes arbitrary integer time units and messaging delay takes one time unit [12, 11], as opposed to real asynchronous models, where local computation takes zero time and the message transit time can be any *real number*. As mentioned before, if the local computation can not be performed in one step, severe issues may occur. These asynchronous P systems assume integer time bound for message transmission, so they do not support theoretical *fairness* that is required by distributed algorithm theory: a message sent is guaranteed to arrive, but there is no time bound for this.

Some asynchronous P systems just apply rules in an asynchronous way: in a step, a cell (i) applies applicable rules, (ii) does not apply applicable rules or (iii) applies an arbitrary number of applicable rules, which do not address real asynchronous concepts [25, 37, 68, 9, 10, 54].

Distributed P systems (dP systems): dP systems split problems in parts, distribute them into various components and construct the solution through the cooperation of these components, which model well-balanced fork/join patterns. However, they are not concurrent models because there are no interactions between the parallel tasks, and they do not address any coordination issues, which are specific to distributed algorithms.

In order to enhance the existing P systems framework for the purpose of modelling complex distributed problems, we define our novel xP systems. As mentioned in the abstract, we aim to answer the following main questions for xP systems, by modelling fundamental and even challenging distributed algorithms.

- Can we construct xP specifications comparable with the corresponding distributed algorithms, based on the following two criteria?
  - Runtime complexity: can xP specifications achieve the same runtime complexities as the corresponding distributed algorithms?

  - Program size: can xP specifications be comparable in program size with the high-level informal non-executable pseudocodes of the corresponding distributed algorithms?

Besides the above main questions, this thesis also aims to provide answers to the following questions.

- Can xP systems relate closely to the synchronous and asynchronous models used in distributed algorithms [62, 42]?
- Can xP systems conform to the usual activation assumptions in the distributed computing framework?
- Can xP systems model complex algorithms that typically need complex data structures?
- Can xP systems enable separation of concern (SoC) designs for complex problems?
- Can xP systems solve problems with fixed-size alphabets and rulesets?

Our coherent modelling approaches give positive answers to all above questions and provide useful experience to the global P systems community for developing P solutions of complex distributed algorithms.

1.4 Thesis Outline

An outline of the thesis is given as follows:

- **Chapter 2:** This chapter defines xP systems, which are extended versions of simple P systems proposed by our joint work [51, 3, 52, 22, 50]. xP systems provide a unified formal model for both synchronous and asynchronous systems: the same static description for both synchronous and asynchronous systems and only the messaging delays differ [3]. Several extended features of xP systems are presented: λ messaging (a new contribution in this thesis), matrix structured rulesets [22], complex symbols [52, 22], cell IDs [51, 52], generic rules [52, 22], complex states [50] and rule fragments composition, including sequential composition (a new contribution in this thesis), parallel composition with no interaction [50] and parallel composition with interaction (a new contribution in this thesis). These extensions provide useful ingredients for modelling fundamental and challenging distributed algorithms. Finally we compare our xP systems with other computational models and summarise the features of xP systems.

- **Chapter 3:** This chapter first provides xP solutions of several fundamental traversal algorithms: Echo, distributed depth-first search (DFS), distributed breadth-first search (BFS) and neighbour discovery algorithms, based on our joint work [3]. As new contributions in this thesis, we address the common problem in distributed algorithms, termination detection problem. We develop xP solutions of several well-known termination detection algorithms and their applications and then obtain complete xP solutions of traversal algorithms, which include both neighbour discovery and termination detection. Finally, we compare our xP specifications and pseudocodes presented in Tel's book [62] in terms of runtime complexity and program size.

- **Chapter 4:** This chapter discusses a series of distributed synchronous depth-first search (DFS) and breadth-first search (BFS) based algorithms, which identify the maximum cardinality set of edge- and node-disjoint paths between a source node and a target node in a digraph. We first review the classical algorithm based on Ford-Fulkerson's maximum flow algorithm [24] and Dinneen et al.'s [18] proposal. Then, following our joint work [51, 3, 52, 22] and my own work [65], we present a set of improved algorithms with informal sequentialised and structural parallel pseudocodes and their directly executable formal xP specifications. We analyse the asymptotic runtime complexities of these algorithms and experimentally benchmark them by using their xP specifications. Finally, we compare our xP specifications and pseudocodes in terms of runtime complexity and program size.

- **Chapter 5:** Following my own work [64], this chapter gives the first synchronous P solution for one of the most challenging problems in distributed computing, which finds a minimum spanning tree (MST) in a weighted undirected graph. This chapter focuses on the synchronous version of the GHS algorithms, SynchGHS [62, 42]. We describe this algorithm using informal structural parallel pseudocodes and discuss synchronisation barriers, which ensure that each synchronised level runs in $3n$ steps; then we present our xP specification with all executable details needed. Finally, we compare our xP specification and pseudocodes in terms of runtime complexity and program size.

- **Chapter 6:** This chapter concludes this thesis and discusses possible future work.

1.5 Publications

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

2. Nicolescu, R., Wu, H.: BFS solution for disjoint paths in P systems. In: Calude, C., Kari, J., Petre, I., Rozenberg, G. (eds.) Unconventional Computation,

Lecture Notes in Computer Science, vol. 6714, pp. 164–176. Springer Berlin Heidelberg (2011)

3. Nicolescu, R., Wu, H.: New solutions for disjoint paths in P systems. Natural Computing 11, 637–651 (2012)

4. Wu, H.: Minimum spanning tree in P systems. In: Pan, L., Păun, G., Song, T. (eds.) Proceedings of the Asian Conference on Membrane Computing (ACMC2012), Huazhong University of Science and Technology, October 15-18, 2012, Wuhan, China. pp. 88–104 (2012)

5. ElGindy, H., Nicolescu, R., Wu, H.: Fast distributed DFS solutions for edge-disjoint paths in digraphs. In: Csuhaj-Varj, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, G. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 7762, pp. 173–194. Springer Berlin Heidelberg (2013)

6. Wu, H.: Fast distributed BFS solution for edge-disjoint paths. In: Yin, Z., Pan, L., Fang, X. (eds.) Proceedings of The Eighth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), 2013, Advances in Intelligent Systems and Computing, vol. 212, pp. 1003–1011. Springer Berlin Heidelberg (2013)

7. Nicolescu, R., Ipate, F., Wu, H.: Towards high-level P systems programming using complex objects. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Y. (eds.) 14th International Conference on Membrane Computing (CMC14), Chisinau, Republic of Moldova, August 20-23, 2013, Proceedings. pp. 255–276 (2013)

Chapter 2

xP Systems

P systems are parallel and distributed computational models, which have been developed rapidly during recent years. Although a number of variants of synchronous and asynchronous P systems have been proposed, many proposals do not seem to provide sufficient ingredients to solve complex distributed computing problems.

Typically, P systems need to relate to the classical synchronous and asynchronous models used in distributed algorithms [62, 42]. P systems may need to conform to activation assumptions (§ 1.1) in the usual distributed computing framework. Distributed algorithms runtime measures only count message transmission time, ignoring local computation that takes zero time. Complex distributed algorithms require complex data structures. Complex distributed problems need fixed-size solutions, independent of the problem size. Furthermore, complex distributed problems may need a higher level design, which divides a problem into smaller problems such that each smaller problem addresses a separate concern.

This chapter presents *xP systems*, as a unified *formal* model for both synchronous and asynchronous systems, which are extended versions of simple P systems, sharing similar features as previous proposal P systems in our joint work [51, 3, 52, 22, 50]. Our xP systems incorporate adequate extensions to achieve the above goals and are used for modelling fundamental and challenging distributed algorithms in this thesis.

This chapter is organised as follows. Section 2.2 describes the formal definition of simple P systems in our joint work [51, 3, 52]. Section 2.3 describes several extended features of xP systems [51, 52, 22, 50]. Section 2.4 compares xP systems with other computational models. Section 2.5 gives a summary of xP systems.

2.1 Transition P Systems

Although many variants of P system have been proposed, most of them are derived or inspired from *transition P systems*, which were introduced by Păun [55].

Definition 2.1 (Transition P Systems). A transition P system is a system $\Pi = (O, C, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, i_0)$, where

- O is the finite non-empty set of symbols;
- $C \subset O$ is the set of *catalysts*;
- μ is a membrane structure, a *rooted tree* consisting of n cells, labelled as $\sigma_1, \sigma_2, \ldots, \sigma_n$;
- $w_i \in O^*$, where $1 \leq i \leq n$, is a *multiset of symbols* (content) in cell σ_i;
- R_i is a finite set of *rewriting rules* associated with cell σ_i;
- i_0 is the label of cell σ_i, which is the output cell of the system, i.e. a cell that can send multisets to the environment.

Note that the membrane structure of a transition P system is a rooted tree and its structural edges between cells function as *duplex* channels.

Rule form

The general form of a rule, $r \in R_i$ is:

$$r : u \to v, \text{ where } u \in O^+ \text{ and } v \in (O \times Tar)^*,$$

where u and v are multisets of symbols and $Tar = \{here, in, out\}$ indicates the *transfer operator*:

- *here*—remains in the same cell;
- *in*—to a nondeterministically chosen child;
- *out*—to the parent.

Symbols of u are called *left-side* symbols and symbols of v are called *right-side* symbols.

Rule applicability

For cell σ_i, a rule, r, is *applicable* if $u \subseteq w_i$.

Rule application

In a step, each cell

1. (a) finds a maximal applicable set of rules, by assigning multiplicities to rules, with two properties: (i) the multiset of (nondeterministically) chosen rules is applicable to the available symbols; and (ii) the multiset is maximal because of the lack of available symbols; and (b) consumes left-side symbols of the rules;

2. *at the end of the step*, produces the right-side symbols of the rules and sends these symbols as indicated by the transfer operators [55].

Example 2.1. Consider the following set of rules, in a system where cell σ_1 contains a^3b^2.

$$r_1\colon\ ab \ \rightarrow\ c$$
$$r_2\colon\ a \ \rightarrow\ d$$
$$r_3\colon\ c \ \rightarrow\ e$$

Case 1:

- Find a maximal set of rules, $\{r_1, r_2\}$, where r_1 is assigned with multiplicity of 2 and r_2 is assigned with multiplicity of 1.
- Rules r_1 and r_2 are applied in parallel: for rule r_1, two copies of a and two copies of b are consumed; for rule r_2, one copy of a is consumed.
- At the end of the step: σ_1 contains dc^2.

Case 2:

- Find a maximal set of rules, $\{r_2\}$, where r_2 is assigned with multiplicity of 3.
- Rule r_2 is applied: three copies of a are consumed.
- At the end of the step: σ_1 contains b^2d^3.

Case 3:

- Find a maximal set of rules, $\{r_1, r_2\}$, where r_1 is assigned with multiplicity of 1 and r_2 is assigned with multiplicity of 2.
- Rules r_1 and r_2 are applied in parallel: for rule r_1, one copy of a and one copy of b are consumed; for rule r_2, two copies of a are consumed.
- At the end of the step: σ_1 contains d^2c.

Configuration

The multisets of symbols in each cell identify a *configuration* of a P system. The *initial configuration* is the configuration at the beginning of a computation, i.e. $(w_1, \ldots, w_i, \ldots, w_n)$, where w_i is the content cell σ_i, $1 \leq i \leq n$. A *transition* is the transformation from a configuration to the next configuration. A sequence of transitions constitutes a *computation*, starting from an *initial configuration* when the

system starts evolution and ending in a *final configuration* when the system terminates.

There are a number of extended features and variants introduced in P systems.

Structure generalisation and communication channels

The membrane structure of a P system can be a *digraph* (neural P systems [56] and tissue P systems [44]), where its structural digraph arcs between cells function as *simplex* channels, i.e. given a structural arc, (σ_i, σ_j), σ_i can send messages to σ_j but vise versa is not true.

The membrane structure can also be a *directed acyclic graph* (hyperdag P systems [49]), where its structural dag arcs between cells function as *duplex* channels, i.e. given a structural dag arc, (σ_i, σ_j), σ_i can send messages to σ_j and vise versa.

Priority

In order to control the use of rules, a *priority* order is considered on the set of rules.

- *Strong priority:* if a higher priority rule is applied, then a lower priority rule can not be applied at all.
- *Weak priority:* a lower priority rule can be applied after all higher priority rules have been applied.

Example 2.2. Consider the same set of rules in Example 2.1 with strong priorities, in a system where cell σ_1 contains a^3b^2.

$$r_1\colon\ ab \ \rightarrow\ c$$

$$r_2\colon\ a \ \rightarrow\ d$$

$$r_3\colon\ c \ \rightarrow\ e$$

- First, rule r_1 is applied: two copies of a and two copies of b are consumed.
- We do not apply any other rules (for the remaining a).
- At the end of the step: σ_1 contains ac^2.

Example 2.3. Consider the same set of rules in Example 2.2 with weak priorities, in a system where cell σ_1 contains a^3b^2.

- First, rule r_1 is applied: two copies of a and two copies of b are consumed.

- Then, rule r_2 is applied (for the remaining a): one copy of a is consumed.
- Rule r_3 is not applicable.
- At the end of the step: σ_1 contains dc^2.

Promoters and inhibitors

Another way to control the use of rules is using *promoters* and *inhibitors*. The *rule form* using promoters is $u \to v \mid z$, where z is a multiset of promoters; this rule is applicable in cell σ_i if $uz \subseteq w_i$, i.e. if all symbols of u and z are available. The rule form using inhibitors is $u \to v \neg z'$, where z' is a multiset of inhibitors; this rule is applicable in cell σ_i if $u \subseteq w_i$ and $z' \cap w_i = \emptyset$, i.e. if all symbols of u are available and none of the symbols of z' is available. Note that, when the rule is applied, promoters and inhibitors are not consumed.

Rewriting mode

There can be several rewriting modes, α, where $\alpha \in \{\texttt{min}, \texttt{par}, \texttt{max}\}$, for applying rules. In the minimal mode, an applicable rule is chosen and applied once. In the parallel mode, an applicable rule is chosen and applied as many times as possible. In the maximal mode, we finds a maximally applicable set of rules and each of these rules is applied as many times as possible.

In our previous proposal simple P systems [51, 3, 52], we specify a rewriting mode, α, where $\alpha \in \{\texttt{min}, \texttt{max}\}$, for each rule, rather than for the whole system (as in transition P systems). Example 2.5 shows how rules are applied according to its rewriting mode.

Example 2.4.

Consider the following set of rules, in a system where cell σ_1 contains a^3b^2.

$$r_1\colon\ ab \to_{\texttt{max}} c$$

$$r_2\colon\ a \to_{\texttt{min}} d$$

$$r_3\colon\ c \to_{\texttt{max}} e$$

Case 1:

- First, rule r_1 is non-deterministically chosen and applied as many times as possible: two copies of a and two copies of b are consumed.
- Rule r_2 is non-deterministically chosen and applied once: one copy of a is consumed.
- Rue r_3 is not applicable because there is no c in the current content.

- At the end of the step: σ_1 contains dc^2.

Case 2:

- First, rule r_2 is non-deterministically chosen and applied once: one copy of a is consumed.
- Rule r_1 is non-deterministically chosen and applied as many times as possible: two copies of a and two copies of b are consumed.
- At the end of the step: σ_1 contains dc^2.

Rule states

The use of states in rules was introduced by tissue P systems [44] and neural P systems [56], which use the rule form $r : S\ u \to S'\ v$, where S is rule r's starting state and S' is the target state. For cell σ_i in state S_i with content w_i, rule r is applicable only if

$$S = S_i, w \subseteq w_i,$$

and

- either no other rule was previously applied in the same step; or
- all rules previously applied in the same step have indicated the same target state, S'.

After rule r is applied, cell σ_i transforms its target state to S'.

Example 2.5.

Consider the following set of rules, in a system where cell σ_1 in state S_0 contains a^3b^2.

$$r_1\colon\ S_0\ ab\ \to S_1\ c$$

$$r_2\colon\ S_0\ a\ \to S_2\ d$$

$$r_3\colon\ S_1\ c\ \to S_2\ e$$

Case 1:

- First, rule r_1 is non-deterministically chosen and applied: two copies of a, two copies of b are consumed and the target state is fixed to S_1.
- Rule r_2 is not applicable because it indicates a target state, S_2, different from the one already selected, S_1.

- Rule r_3 is not applicable, for two reasons: (1) its starting state, S_2, is different from σ_1's current state, S_0; and (2) there is no c in the current content.
- At the end of the step: σ_1 contains ac^2 and enters state S_1.

Case 2:

- First, rule r_2 is non-deterministically chosen and applied: three copies of a are consumed and the target state is fixed to S_2.
- We can not apply any other rules (because of the lack of available symbols).
- At the end of the step: σ_1 contains b^2d^3 and enters state S_2.

Simple P systems [51, 3, 52] leverage several of the above features: a digraph membrane structure with duplex channels, weak priority of rules, promoters, inhibitors, rule states and a rewriting mode $\{\texttt{min}, \texttt{max}\}$ for each rule.

2.2 Simple P Systems

Traditional P systems are synchronous, i.e. all cells evolve under the control of a single global clock. For asynchronous P systems, there is no global clock; various asynchronous features have been investigated by recent research [6, 12, 9, 11, 10, 25, 37, 54, 68, 3].

Following our joint work [51, 3, 52], this chapter describes a *simple P system*, which is a unified formal model for both synchronous and asynchronous systems. Simple P systems use the same *static* description for both synchronous and asynchronous P systems and only their *message transmission* differs [3]: in asynchronous P systems, messages are transferred asynchronously, while in synchronous P systems, messages are transferred synchronously. Simple P systems use the same runtime and message complexity measures discussed in Section 1.2.

Definition 2.2 (Simple P systems). A simple P system is a system $\Pi = (V, E, Q, O, R)$, where

- V is a finite set of cells;
- E is a set of structural *parent-child digraph* arcs between cells, functioning as *duplex* channels;
- Q is a finite set of states;
- O is a finite non-empty alphabet of symbols; and
- R is a finite set of multiset rewriting rules.

All components of a simple P system, i.e. V, E, Q, O and R, are *immutable*.

Distributed solution

In this thesis, a distributed solution is provided as a P system itself: the graph or digraph of the problem is represented as the underlying structure of a P system, not by encoding.

A graph, $G_1 = (V, E_1)$, can be represented by the underlying structure of a P system, $G = (V, E)$, using one of the following ways. In this thesis, we use approach (1).

1. edge $\{u, v\} \in E_1$ is (arbitrarily) represented by arc (u, v) or (v, u), i.e. $(u, v) \in E$ or $(v, u) \in E$, with a duplex channel, as shown in Figure 2.1 (a);
2. edge $\{u, v\} \in E_1$ is represented by arc pair (u, v) and (v, u), i.e. $\{(u, v), (v, u)\} \subset E$, with simplex channels of the same direction as the arcs, as shown in Figure 2.1 (b).

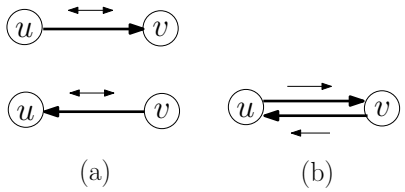


Figure 2.1: How to represent an edge of a graph in a P system. Thick arcs: structural arcs; arrows beside arcs: channel directions.

A digraph, $G_2 = (V, E_2)$, can be represented by exactly the *same* underlying P system structure, $G = (V, E)$: arc $(u, v) \in E_2$ is represented by arc $(u, v) \in E$, i.e. $E_2 = E$, with a duplex channel, as shown in Figure 2.2 (a), or a simplex channel of the same direction as the arc, as shown in Figure 2.2 (b). In this thesis, we assume duplex channels in all digraphs.

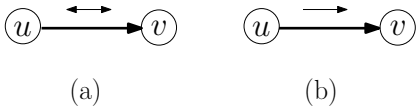


Figure 2.2: How to represent an arc of a digraph in a P system. Thick arcs: structural arcs; arrows beside arcs: channel directions.

All cells start in the same state with the same ruleset; cells are only differentiated by the initial content and cell IDs (later discussed in Section 2.3.4); cells may need initial differentiation to enter their corresponding states for further processing (as in Chapter 4); cells may or may not know the identities of their neighbours or their numbers.

Configuration

Each cell, $\sigma_i \in V$, has the *initial configuration* (S_{i0}, w_{i0}), and the *current configuration* (S_i, w_i), where $S_{i0} \in Q$ is the initial state; $S_i \in Q$ is the current state; $w_{i0} \in O^*$ is the initial multiset of symbols; and $w_i \in O^*$ is the current multiset of symbols. A *configuration* of a simple P system consists of current configurations of all cells.

Rule form

The general form of a rule, $r_j \in R$, is:

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

where

- $S, S' \in Q$, S is rule r_j's *starting* state and S' is r_j's *target* state;
- $x, x', y', y, z, z' \in O^*$;
- $\alpha \in \{\texttt{min}, \texttt{max}\}$ is a *rewriting mode*;
- β_γ is a *transfer operator*, $\beta \in \{\uparrow, \downarrow, \updownarrow\}$ and $\gamma \in V \cup \{\forall\}$;
- ellipses $(\ldots)$ indicate possible repetitions of the last parenthesised item.

Rule applicability

The rules given by the ruleset $R = (r_1, \ldots r_m)$ are applied in the *weak* priority order. For cell σ_i in configuration (S_i, w_i), a rule, r_j, is *applicable* if

$$S = S_i,\ xz \subseteq w_i,\ z' \cap w_i = \emptyset,$$

where multiset z is a *promoter* and z' is an *inhibitor*, which enables and disables the rule respectively, without being consumed [57] and

(a) either no other rule was previously applied in the same step; or

(b) all rules previously applied in the same step have indicated the same target state, S'.

Pseudocode 2.1 shows how ruleset R is applied.

Pseudocode 2.1: Ruleset application

2.1.1 **Input** : a simple P system, $\Pi = (V, E, Q, O, R), R = (r_1, \ldots r_m), 1 \leq m$
2.1.2 **for** $j = 1$ **to** m
2.1.3 **if** r_j is applicable **then**
apply r_j: *multiset x' becomes immediately available*
messages y' and y are transferred with arbitrary delays
endif
2.1.4 **endfor**

Rule application

After rule r_j is *applied*:

(1) the target state is fixed to S', if not already fixed;

(2) multiset x is consumed;

(3) multiset x' becomes *immediately available* in the same cell (this is an extension, matrix structured rulesets, proposed by our joint work [22]);

(4) message y' is sent to the same cell via a *loopback* channel;

(5) message y is sent as indicated by the transfer operator β_γ.

— β indicates the *transfer operator*:
· $\uparrow$—to parents;
· $\downarrow$—to children;
· $\updownarrow$—in both directions.
— γ indicates the *distribution form*:
· $\forall$—a broadcast, which is the default distribution form if no γ is specified,
· a *structural neighbour*, $\sigma_j \in V$—a unicast (to this neighbour).

Operator α describes the rewriting mode:

- `min` indicates that r_j is applied once;
- `max` indicates that r_j is applied as many times as possible.

Symbols of x are called *left-side* symbols; symbols of x' and y' are called *here* symbols; and symbols of y are called *message* symbols.

We propose *hidden loopback* channels for each cell, $\{(v, v) \mid v \in V\}$, and make a fine distinction between two kinds of "here" symbols: (1) symbols of x' become

immediately available for lower priority rules [22]; (2) symbols of y' can be thought, as traditionally in P systems, as being messaged back to the same cell via a hidden loopback channel, and thus cannot be used by lower priority rules.

Classical P systems' treating "here" symbols as loopback messages can transform even simple local computations into complex computations running in several steps, which does not match the common runtime complexity's assumption that local computation takes zero time (§ 1.1).

Having "here" symbols immediately available after being generated provides a facility for completing local computations in zero time, which conforms to the assumption of common runtime complexity (§ 1.2). As later discussed in Section 2.3.5, by using *generic rules* and *complex symbols*, local computations such as determining the minimum of an arbitrary long sequence can be computed in one single step, i.e. zero time [22].

Example 2.6 explains how a set of rules are considered for applicability and applied in one step. To simplify the explanations, in this chapter, examples are illustrated with *synchronous* P systems, where message transmission delay is exactly one time unit; for asynchronous simple P systems, the rule applications are the same: only transmission delay differs.

Example 2.6. Consider the following rules with weak priorities, $\{r_1, r_2, r_3\}$, in a *synchronous* system where cell σ_1 contains one symbol, a, and has one child cell, σ_2.

$$
\begin{aligned}
r_1\colon\ & S_0\ a\ \rightarrow_{\texttt{min}}\ S_0\ c\ (b)\ (f)\downarrow_2 \\
r_2\colon\ & S_0\ b\ \rightarrow_{\texttt{min}}\ S_1\ d\ (g)\downarrow_2 \\
r_3\colon\ & S_0\ c\ \rightarrow_{\texttt{min}}\ S_0\ a\ (h)\downarrow_2
\end{aligned}
$$

- First, rule r_1 is applied: one c becomes immediately available (which can be used by lower priority rules); one b is sent to itself; and one f is sent to σ_2. Also, the target state is fixed to S_0.
- Next, rule r_2 is not applicable, for two distinct reasons: (1) there is no b in the current content (the message b sent to itself by rule r_1 takes one time unit to transfer) and (2) it indicates a target state, S_1, different from the one already selected, S_0.
- Finally, rule r_3 is applied: one a becomes available and one h is sent to σ_2.
- After one time unit, σ_1 contains ab and message fh arrives at σ_2.
- All applicable rules, r_1 and r_3, are applied in *one* step.

2.3 Extensions

Following our joint work [51, 52, 22, 50], this section provides several extensions: λ messaging (a new contribution in this thesis) makes P systems conform to usual activation assumptions (§ 1.1); matrix structured rulesets [22] provide ingredients to achieve the same runtime complexity as in distributed computing framework; complex symbols [52, 22] and cell IDs provide [51, 52] complex data structures needed by complex algorithms, to achieve fixed-size alphabets and rulesets; generic rules process complex symbols and also provide ingredients to achieve the same runtime complexity as in distributed computing framework; complex states and rule fragments composition, including sequential composition (a new contribution in this thesis), parallel composition with no interaction [50] and parallel composition with interaction (a new contribution in this thesis), allow separation of concern (SoC) designs for complex problems. The extended simple P systems are hereafter called *xP systems*.

2.3.1 λ messaging

In classical P systems, a rule can be applicable at the start of next step without receiving a message, which does not conform to activation assumptions (§ 1.1) in the usual distributed computing framework.

To conform to activation assumptions, we propose a semantic extension of rule processing. In principle, when a cell applies a rule, it sends a λ message to itself via a loopback channel. This ensures that the cell becomes active at the start of the next step by receiving a message. However, we have one optimization: if this rule application does not make any rule applicable at the start of the next step (without receiving another message), then no λ message needs to be sent. Note that, this λ message is assumed to be automatically sent and does not need to explicitly appear in any rule.

With this extension, all xP systems conform to the usual activation assumptions.

Example 2.7. In Example 2.6, one a is immediately available after rule r_3 is applied and makes rule r_1 applicable at the start of the next step without receiving a message. Thus, when cell σ_1 applies rule r_3, it also sends a λ loopback message, to ensure that σ_1 becomes active at the start of next step by receiving a message.

However, if we change rule r_3 to r'_3 by replacing a with (a), which is treated as a loopback message, then no λ message needs to be sent because the application of rule r'_3 does not make any rule applicable at the start of the next step without receiving a message.

$$
\begin{aligned}
r_1\colon\ & S_0\ a\ \rightarrow_{\texttt{min}}\ S_0\ c\ (b)\ (f)\downarrow_2\\
r_2\colon\ & S_0\ b\ \rightarrow_{\texttt{min}}\ S_1\ d\ (g)\downarrow_2\\
r'_3\colon\ & S_0\ c\ \rightarrow_{\texttt{min}}\ S_0\ \boldsymbol{(a)}\ (h)\downarrow_2
\end{aligned}
$$

2.3.2 Matrix Structured Rulesets

Matrix structured rulesets are proposed by our joint work [22], which are inspired by matrix grammars with appearance checking [26]. For matrix grammar with appearance checking, in a step, *several* vectors are applied in *weak* priority; in a vector, non-skipped rules (rules can be skipped with appearance checking) are applied in *weak* priority. For our matrix structured rulesets, in a step, only one vector is applied in *strong* priority; in a vector, applicable rules are applied in *weak* priority.

Matrix structured rulesets combine (1) a *strong priority* for vectors and (2) a version of *weak priority* for rules inside a vector. Ruleset R is organised as a *matrix*, i.e. a list of *vectors*, where vectors are listed from high to low priorities:

$$R = (R_1, \ldots, R_m), 1 \leq m$$

Each vector R_i is a sequence of rules, where rules are listed from high to low priorities, sharing the *same starting state*:

$$R_i = (R_{i,1}, \ldots, R_{i,m_i}), 1 \leq m_i$$

Pseudocode 2.2 shows how ruleset R is applied. Note that, in a matrix structured ruleset, a rule of vector R_i is denoted by $R_{i,j}$, where $1 \leq i \leq m$ and $1 \leq j \leq m_i$.

Pseudocode 2.2: Matrix structured ruleset application

```
2.2.1 Input : an xP system, Π = (V, E, Q, O, R)
2.2.2         R = (R_1, ... R_m), 1 ≤ m, and R_i = (R_{i,1}, ..., R_{i,m_i}), 1 ≤ m_i
2.2.3 applied = false
2.2.4 for i = 1 to m
2.2.5   for j = 1 to m_i
2.2.6     if R_{i,j} is applicable then
            apply R_{i,j}: multiset x' becomes immediately available
                           messages y' and y are transferred with arbitrary delays
            applied = true
          endif
2.2.7   endfor
2.2.8   if applied then break
2.2.9 endfor
```

Rule form

The rule form is the same as discussed in Section 2.2.

Rule applicability

In a given vector, R_i, rules are considered for application according to their weak priority order:

(a) if applicable, a higher priority rule is applied before considering the next lower priority rule;

(b) otherwise (if not applicable), a higher priority rule is silently ignored and the next lower priority rule is considered.

Rule application

After a rule is applied, multiset x' becomes immediately available for lower priority rules in the same vector; messages y' and y are transferred with arbitrary transmission delays, which cannot be used by lower priority rules in the same vector.

Vector applicability

A vector is applicable if at least one of its rules is applicable. Vectors are considered for application in their strong priority order:

(a) if applicable, a higher priority vector is applied in *one single* step and all lower priority vectors are ignored (for the current step);

(b) otherwise (if not applicable), a higher priority vector is silently ignored and the next lower priority vector is considered.

Matrix structured rulesets also provide an ingredient to ensure that the local computation implemented by a vector is performed in one single step, i.e. zero time.

2.3.3 Complex Symbols

While elementary symbols seem sufficient for many theoretical studies (e.g, computational completeness), complex algorithms need adequate complex data structures. Our joint work [52, 22] enhances the initial vocabulary by recursive composition of *elementary symbols* from O into *complex symbols*, which can be viewed as complex molecules, consisting of elementary atoms or other molecules. Complex symbols are Prolog-like [15] compound terms of the form:

$$t(i, \dots),$$

where

1. t is an *elementary symbol* representing the functor (like the functor in Prolog);
2. i can be
   (a) an *elementary* symbol,
   (b) another *complex* symbol,

(c) a *free variable* (open to be bound, according to the cell's current configuration),

(d) a *multiset* of elementary and complex symbols and free variables.

Free variables are used for *pattern matching* on term arguments and typically denoted by lowercase subscripts such as i, j, k, or uppercase letters such as X, Y, Z.

Example 2.8. The following are examples of complex symbols, where a, b, c, d, e, f are elementary symbols and i, j, X are free variables (assuming that these are not listed among elementary symbols):

- $a(i) = a_i$,
- $b(2) = b_2$,
- $c(), c(c()) = c^2(), c(c(c())) = c^3(), \ldots,$
- $d(i, j) = d_{i,j}$,
- $e(a^2b^3)$,
- $f(j, c^5) = f_j(c^5)$,
- $f(j, X) = f_j(X)$.

Note that the counters, $c(), c^2(), c^3(), \ldots$, can be interpreted as integers, 0, 1, 2, ..., which can be used for representing and processing any number of cell IDs with a fixed vocabulary (as later discussed in the Section 2.3.4)

Complex symbols provide ingredients to achieve smaller program size than equivalent P solutions using elementary symbols. As later seen in Section 3.2, an xP solution has less than half of the rules of a P solution proposed by Kim [35], which uses elementary symbols.

Complex symbols can be useful for modelling complex data structures, such as lists, stacks, trees and dictionaries, or emulating procedure calls [50].

2.3.4 Cell IDs

To allow the same ruleset for all cells, cells should only differ in their initial contents and relative positions in the underlying digraph. Thus, it is necessary to let each cell keep its own unique *cell ID*.

Following our joint work [51, 52], each cell σ_i is "blessed" with a unique complex *cell ID* symbol, $\iota(i)$, typically abbreviated as ι_i. The cell ID symbol, ι_i, is *accessible* to the rules, but is exclusively used as an *immutable promoter*.

Note that, the introduction of cell IDs does *not* increase the alphabet size [48]: each cell ID can be encoded with a fixed number of elementary symbols and thus it is part of the alphabet.

One of the possible representations of cell IDs is using the counters discussed in Example 2.8, e.g., a cell ID 3 can be represented by complex symbol c(c(c(c()))); other representations can also be considered, such as unary or binary strings [48].

2.3.5 Generic Rules

To process complex symbols, our joint work [52, 22] proposes high-level generic rules, which are identified by an extended version of the classical rewriting mode, a combined *instantiation.rewriting* mode, where (1) the *instantiation* mode is one of {`min`, `max`} and (2) the *rewriting* mode is one of {`min`, `max`}. Four combinations of the instantiation and rewriting modes are used: `min.min`, `min.max`, `max.min`, `max.max`.

The instantiation and application of a generic rule are explained as follows.

- The instantiation mode indicates how many instance rules are conceptually generated, using *free variable* matching:
  - `min` indicates that the generic rule is nondeterministically generated only *once*, if possible;
  - `max` indicates that the generic rule is repeatedly generated as *many* times as possible, depending on the actually cell contents, without superfluous instances (i.e. without duplicates, see Example 2.9 (c)).

  Note that the rule instantiation is based on the actual cell content and thus a generated rule is *always applicable* and applied according to the rewriting mode.
- The rewriting mode indicates how each instantiated rule is applied (as in the classical framework, which is the same as discussed in Section 2.2).
  - `min` indicates that the instantiated rule is applied once;
  - `max` indicates that the instantiated rule is applied as many times as possible.

  After the instantiated rule is applied, if the instantiation mode is `max`, then the generic rule repeats the generation process until no new rules can be generated.

Pseudocode 2.3 shows how generic rule $R_{i,j}$ is instantiated and applied, which can replace the codes in boxed area in Pseudocode 2.2, if $R_{i,j}$ is a generic rule.

Pseudocode 2.3: Generic rule instantiation and application

2.3.1 **Input** : a generic rule, $R_{i,j} \in R_i$, where $R_i \in R$, of the form
2.3.2 $S\ x\ \rightarrow_\alpha\ S'\ x'\ (y')\ (y)\beta_\gamma \ldots \mid z \neg z'$,
2.3.3 where $\alpha \in \{\texttt{min.min}, \texttt{min.max}, \texttt{max.min}, \texttt{max.max}\}$
2.3.4 **while** ($R_{i,j}$ can generate a new rule)

2.3.5 **instantiate** $R_{i,j}$
2.3.6 apply $R_{i,j}$: *multiset x' becomes immediately available*
2.3.7 *messages y' and y are transferred with arbitrary delays*
2.3.8 applied = **true**
2.3.9 **if** $\alpha = \texttt{min.min}$ **or** $\texttt{min.max}$ **break**
2.3.10 **endwhile**

Example 2.9. Consider a system where cell σ_7 contains multiset $f_2 {f_3}^2 v$, and the generic rule ρ_α, where $\alpha \in \{\texttt{min.min}, \texttt{min.max}, \texttt{max.min}, \texttt{max.max}\}$ and i and j are free variables [22]:

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

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

$$(\rho'_1)\ S_{20}\ f_2\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_2 \mid v\ \iota_7$$
$$(\rho''_1)\ S_{20}\ f_3\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_3 \mid v\ \iota_7$$

In the first case, using (ρ'_1), cell σ_7 ends with $f_3^2 v$.
In the second case, using (ρ''_1), cell σ_7 ends with $f_2 f_3 v$.

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

$$(\rho'_2)\ S_{20}\ f_2\ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_2 \mid v\ \iota_7$$
$$(\rho''_2)\ S_{20}\ f_3\ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_3 \mid v\ \iota_7$$

In the first case, using (ρ'_2), cell σ_7 ends with $f_3^2 v$.
In the second case, using (ρ''_2), cell σ_7 ends with $f_2 v$.

(c) $\rho_{\texttt{max.min}}$ nondeterministically generates one of the following *lists* of rule instances (each list is an ordered sequence of rules that are applied in weak priority):

$$(\rho'_3)\ S_{20}\ f_2\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_2 \mid v\ \iota_7$$
$$(\rho''_3)\ S_{20}\ f_3\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_3 \mid v\ \iota_7$$
$$\ldots$$
$$(\rho'_4)\ S_{20}\ f_3\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_3 \mid v\ \iota_7$$
$$(\rho''_4)\ S_{20}\ f_2\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_2 \mid v\ \iota_7$$

In the first case, using (ρ'_3) and (ρ''_3), cell σ_7 ends with $f_3 v$.
In the second case, using (ρ'_4) and (ρ''_4), cell σ_7 also ends with $f_3 v$.
In both cases, although σ_7 still contains f_3, rule $S_{20}\ f_3\ \rightarrow_{\texttt{min}}\ S_{20}\ (b_7)\updownarrow_3$ can not be generated again because `max` instantiation mode does not allow duplicates.

(d) $\rho_{\texttt{max.max}}$ nondeterministically generates one of the following *lists* of rule instances (each list is an ordered sequence of rules that are applied in weak priority):

$$\begin{array}{ll} (\rho'_5) & S_{20}\ f_2 \ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_2 \mid v\ \iota_7 \\ (\rho''_5) & S_{20}\ f_3 \ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_3 \mid v\ \iota_7 \\ & \ldots \\ (\rho'_6) & S_{20}\ f_3 \ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_3 \mid v\ \iota_7 \\ (\rho''_6) & S_{20}\ f_2 \ \rightarrow_{\texttt{max}}\ S_{20}\ (b_7)\updownarrow_2 \mid v\ \iota_7 \end{array}$$

In the first case, using (ρ'_5) and (ρ''_5), cell σ_7 ends with v.
In the second case, using (ρ'_6) and (ρ''_6), cell σ_7 also ends with v.

Example 2.10. Consider a cell that contains the following list of complex symbols: $m(c^{i_0})$, $a_1(c^{i_1})$, $a_2(c^{i_2})$, ..., $a_n(c^{i_n})$, representing the values i_0, i_1, i_2, ..., i_n, respectively (where $n \geq 0$). The following generic rule, μ, determines the minimum over this sequence of values, in *one single step* [22]:

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

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

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

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

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

In a distributed algorithm, a node can determine the minimum over an arbitrary long local sequence in one single step, which takes zero time. By using `max` instantiation mode of a generic rule, our xP system achieves the same runtime performance.

The instantiation of generic rules is

- *conceptual*: it explains their high-level semantics by mapping it to a simpler lower-level semantics.

- *ephemeral*: the lower-level rules are generated when rules are tested for applicability and are not supposed to exist past the end of the step.

An actual P system implementation *does not* need to effectively use rule instantiation, as long as it can support the same high-level semantics by other means.

Generic rules allow (1) a reasonably fast parsing and processing of complex symbols, and (2) algorithm descriptions with *fixed-size alphabets* and *fixed-size rulesets*, independent of the problem size and number of cells in the system (sometimes impossible with only elementary symbols).

2.3.6 Complex States

We enhance the initial vocabulary from *elementary states* of Q to *complex states*, which are compound terms of the form:

$$\Theta(i_1, i_2, \ldots, i_n), n \geq 1, \text{ where}$$

1. Θ is a functor;
2. each i_j ($1 \leq j \leq n$) is an elementary state or a free variable.

Example 2.11. The following are examples of complex states, where X, Y, Z are free variables:

- $\Theta(S_2) = S_2$,
- $\Theta(X)$,
- $\Theta(S_2, X)$
- $\Theta(Y, S_1)$,
- $\Theta(S_3, X, Y, Z)$.

Free variables are denoted by uppercase letters that are not used as elementary states, such as X, Y, Z. Next, Section 2.3.7 explains in detail how complex states are used and mapped in the parallel system composition.

2.3.7 Rule Fragments Composition

A P solution of a complex distributed problem can be divided into smaller rule fragments and the solution is just the composition of bigger chunks out of the rule fragments. The composition of rule fragments enables separation of concerns (SoC), a well-known design principle for separating a computer program into distinct sections, such that each section addresses a separate concern.

The composition can be *sequential* or *parallel*. Here we present three compositions: sequential composition (a new contribution in this thesis), parallel composition with no interaction [50] and parallel composition with interaction (a new contribution in this thesis).

Consider systems Π_1 and Π_2, which share the *same membrane structure*.

Sequential Composition

Sequential composition can be considered as having each cell apply rules of Π_2 *exclusively* after it finishes applying rules of Π_1. In this thesis, the sequential composition is used for algorithms including a preliminary local topology discovery phase, as in Sections 3.5 and 3.7 and Chapter 4.

Consider Π_1 and Π_2, which satisfy the following condition:

- Π_1 and Π_2 share exactly *one* state, S_c: any rule of Π_1 that uses S_c can only use it as the target state and S_c is the starting state of at least one rule of Π_2.

The sequential composition of Π_1 and Π_2, denoted as $\Pi_1; \Pi_2$, is constructed in the following way:

1. $\Pi_1; \Pi_2$ shares the same structure as Π_1 and Π_2;
2. rules of Π_1 and rules of Π_2 are concatenated, with rules of Π_1 having higher priority than rules of Π_2.

Consider systems Π_1 with m rules and Π_2 with n rules. The composed $\Pi_1; \Pi_2$ has $m + n$ rules (see Example 2.12), where the m rules of Π_1 have higher priority than the n rules of Π_2 and all these rule are considered to be applied in *one* single step.

Typically, for the entire system of cells, Π_1 and Π_2 can run in parallel: some cells apply rules of Π_1 while other cells apply rules of Π_2.

This composition is adequate if each cell applies rules of Π_2 after it finishes applying rules of Π_1, e.g., in the synchronous setting, a cell starts a subsequent phase (Π_2) after its local topology discovery (Π_1). However, if a cell applies rules of Π_2 only after Π_1 terminates, e.g., in the asynchronous setting, a cell starts a subsequent phase (Π_2) after all cells finish their local topology discovery (Π_1), one needs additional constraints such that Π_2 starts only after Π_1 terminates. Example 2.12 shows an example that Π_2 starts only after Π_1 terminates.

Example 2.12.

Consider Π_1 and Π_2 sharing the same membrane structure. Π_1 discovers all σ_s's children and Π_2 notifies all σ_s's children of their siblings. Figures 2.3 and 2.4 show the initial and final configurations of Π_1 and Π_2, respectively.

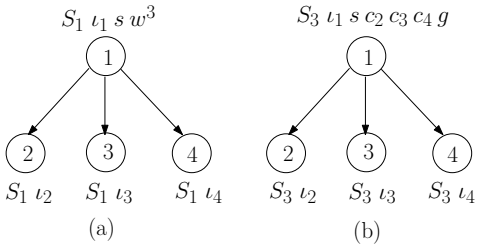


Figure 2.3: (a) the initial configuration of Π_1: the source cell, σ_s, is marked by s and contains a counter, w^3, which indicates σ_s's outdegree; (b) the final configuration of Π_1: σ_s is marked with s and contains children pointers, c_j's, and one g that indicates σ_s knows that Π_1 terminates.

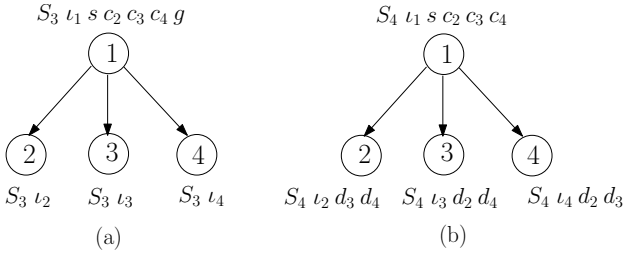


Figure 2.4: (a) the initial configuration of Π_2: the source cell, σ_s, is marked by s and contains one g and children pointers, c_j's; (b) the final configuration of Π_2: σ_s is marked with s and contains children pointers, c_j's; each child of σ_s contains siblings' pointers, d_j's.

Π_1 and Π_2 use the *same single* source cell, σ_1, and share exactly one state, $S_c = S_3$, which is *only* used as the target state in Π_1 (as in rules r_2 and r_4) and used as the starting state of rules r_5–r_8 in Π_2. Symbol g is generated by Π_1, indicating that Π_1 terminates, and is used as a promoter by rule r_5 to start Π_2.

- Π_1, has 3 states and 4 rules:

$$
\begin{array}{lll}
r_1: & S_1 & \rightarrow_{\texttt{min.min}} \quad S_2\ (a_i)\downarrow \mid \iota_i\ s \\
r_2: & S_1\ a_j & \rightarrow_{\texttt{min.min}} \quad S_3\ (b_i)\uparrow_j \mid \iota_i \\
r_3: & S_2\ b_j\ w & \rightarrow_{\texttt{min.min}} \quad S_2\ c_j \mid s \\
r_4: & S_2 & \rightarrow_{\texttt{min}} \quad S_3\ g \mid s\ \neg w
\end{array}
$$

  Cell σ_s starts Π_1 by sending one a_s to each child (rule r_1); each receiving child, σ_i, sends back an acknowledgment, b_i, to σ_s and enters S_3 (rule r_2). For each received b_j, σ_s decrements its counter by one by deleting one w (rule r_3); when σ_s receives all acknowledgments from its children, $\neg\ w$, σ_s generates one g, indicating that it knows that Π_1 terminates, and enters S_3 (rule r_4).

- Π_2, has 2 states and 4 rules:

$$\begin{array}{lll} r_5: & S_3 & \rightarrow_{\texttt{max.min}} \quad S_4\,(c_j)\downarrow \mid s\; g\; c_j \\ r_6: & S_3\; g & \rightarrow_{\texttt{min}} \quad S_4 \\ r_7: & S_3\; c_i & \rightarrow_{\texttt{min.min}} \quad S_4 \mid \iota_i \\ r_8: & S_3\; c_j & \rightarrow_{\texttt{max.min}} \quad S_4\; d_j \end{array}$$

  If cell σ_s contains g, it starts Π_2 by sending all its children pointers, c_j's, to each child (rule r_5); then σ_s erases g and enters S_4 (rule r_6). Each receiving child, σ_i, keeps only its siblings' pointers, by deleting c_i (rule r_7) and transforming c_j's into d_j's, and enters S_4 (rule r_8).

- $\Pi_1; \Pi_2$, has 4 $(= 3 + 2 - 1)$ states and 8 $(= 4 + 4)$ rules:

$$\begin{array}{lll} S_1 & \rightarrow_{\texttt{min.min}} \quad S_2\,(a_i)\downarrow \mid \iota_i\; s \\ S_1\; a_j & \rightarrow_{\texttt{min.min}} \quad S_3\,(b_i)\uparrow_j \mid \iota_i \\ S_2\; b_j\; w & \rightarrow_{\texttt{min.min}} \quad S_2\; c_j \mid s \\ S_2 & \rightarrow_{\texttt{min}} \quad S_3\; g \mid s\; \neg w \\ S_3 & \rightarrow_{\texttt{max.min}} \quad S_4\,(c_j)\downarrow \mid s\; g\; c_j \\ S_3\; g & \rightarrow_{\texttt{min}} \quad S_4 \\ S_3\; c_i & \rightarrow_{\texttt{min.min}} \quad S_4 \mid \iota_i \\ S_3\; c_j & \rightarrow_{\texttt{max.min}} \quad S_4\; d_j \end{array}$$

  $\Pi_1; \Pi_2$ discovers all σ_s's children and notifies σ_s's children of their siblings. Figure 2.3 (a) shows the initial configuration of $\Pi_1; \Pi_2$ and Figure 2.4 (b) shows the final configuration of $\Pi_1; \Pi_2$.

Parallel Composition with No Interaction

Parallel composition with no interaction of two systems, Π_1 and Π_2, can be considered as running in parallel Π_1 and Π_2, which are fully independent: they do not share any symbol or state. This composition is proposed in our joint work [50], which builds the basics for parallel composition with interaction and can be used to model many parallel systems, such as multiple program multiple data (MPMD) [67].

Consider Π_1 and Π_2, which satisfy the following conditions:

- Π_1 and Π_2 use disjoint sets of states (if not, without loss of generality, we can relabel the states to satisfy this condition);
- Π_1 and Π_2 use disjoint sets of symbols.

The parallel composition of Π_1 and Π_2 with no interaction, denoted as $\Pi_1 \parallel \Pi_2$, is constructed in the following way:

1. $\Pi_1 \parallel \Pi_2$ shares the same structure as Π_1 and Π_2;
2. rules of Π_1 and Π_2 are concatenated, with rules of Π_1 having higher priority than rules of Π_2;
3. each state S_i of Π_1 is replaced by complex state $\Theta(S_i, Y)$ and each state S'_i of Π_2 is replaced by complex state $\Theta(X, S'_i)$.

Consider systems Π_1 with m rules and Π_2 with n rules. The composed $\Pi_1 \parallel \Pi_2$ also has $m+n$ rules (see Example 2.13), where the m rules of Π_1 have higher priority than the n rules of Π_2 and all these rule are considered to be applied in *one* single step. Note that, we enforce rules of Π_1 having higher priority than rules of Π_2 in order to have a uniform construction of both parallel composition with no interaction and with interaction. In fact, as later discussed, in $\Pi_1 \parallel \Pi_2$, the priority of rules does not matter because Π_1 and Π_2 do not share any state or symbol.

We denote the rules of Π_1 after replacing states with complex states as $\overline{\Pi_1}$ and the rules of Π_2 after replacing states with complex states as $\overline{\Pi_2}$.

Although we let rules of Π_1 have higher priority than rules of Π_2, actually the priority of rules of $\Pi_1 \parallel \Pi_2$ does not matter because Π_1 and Π_2 use disjoint sets of states and symbols. Thus, rules of $\Pi_1 \parallel \Pi_2$ can be interleaved and this composition is commutative, $\Pi_1 \parallel \Pi_2 = \Pi_2 \parallel \Pi_1$.

Assume that Π_1 uses M states and Π_2 uses N states. By using parallel composition with no interaction, $\Pi_1 \parallel \Pi_2$ uses only $O(M + N)$ rules.

We use *weak binding* for matching variables on components of state symbols: during rules' application, the final target state is successively refined according to the current rule's target state; at the end of the step, the unmapped variables are set according to the cell's current state. Example 2.13 shows an example, which does not do any practically meaningful work, except that Π_1 loops over three states and Π_2 loops over two states.

Example 2.13.

- Π_1, has 3 states and 3 rules:

$$\begin{array}{lll} S_1\, a & \rightarrow_{\texttt{min}} & S_2\, b \\ S_2\, b & \rightarrow_{\texttt{min}} & S_3\, c \\ S_3\, c & \rightarrow_{\texttt{min}} & S_1\, a \end{array}$$

- Π_2, has 2 states and 2 rules:

$$\begin{array}{lll} S'_1\, d & \rightarrow_{\texttt{min}} & S'_2\, e \\ S'_2\, e & \rightarrow_{\texttt{min}} & S'_1\, d \end{array}$$

- $\Pi_1 \parallel \Pi_2$, has 6 $(= 3 \cdot 2)$ states and 5 $(= 3 + 2)$ rules:

$$
\begin{array}{l}
\overline{\Pi_1}: \\
r_1 : \Theta(S_1, Y)\, a \;\; \rightarrow_{\texttt{min}} \;\; \Theta(S_2, Y)\, b \\
r_2 : \Theta(S_2, Y)\, b \;\; \rightarrow_{\texttt{min}} \;\; \Theta(S_3, Y)\, c \\
r_3 : \Theta(S_3, Y)\, c \;\; \rightarrow_{\texttt{min}} \;\; \Theta(S_1, Y)\, a
\end{array}
$$

$$
\begin{array}{l}
\overline{\Pi_2}: \\
r_4 : \Theta(X, S_1')\, d \;\; \rightarrow_{\texttt{min}} \;\; \Theta(X, S_2')\, e \\
r_5 : \Theta(X, S_2')\, e \;\; \rightarrow_{\texttt{min}} \;\; \Theta(X, S_1')\, d
\end{array}
$$

Consider the above rules of $\Pi_1 \parallel \Pi_2$ in a synchronous system where cell σ_1 in state $\Theta(S_1, S_1')$ contains multiset ad. Cell σ_1's target state is mapped during the rules application by weak binding.

1. First, rule r_1 is applied: one a is consumed and one b becomes immediately available; σ_1's target state is temporarily mapped to $\Theta(S_2, Y)$.
2. Next, rule r_4 is applied: one d is consumed and one e becomes immediately available; σ_1's target state is now fixed to $\Theta(S_2, S_2')$.
3. At the end of the step, σ_1 is in state $\Theta(S_2, S_2')$ and contains multiset be.

Note that, if no more rules are applicable after applying rule r_1, at the end of the step, σ_1's unmapped variable, Y, is set according to σ_1's current state to S_1', and thus σ_1's target state is fixed to $\Theta(S_2, S_1')$.

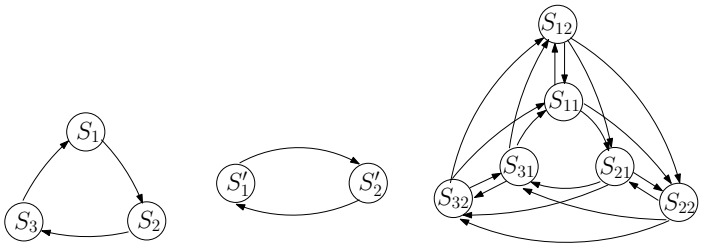
The composition of $\Pi_1 \parallel \Pi_2$ generates $M \cdot N = 6$ states: $\Theta(S_1, S_1')$, $\Theta(S_2, S_1')$, $\Theta(S_3, S_1')$, $\Theta(S_1, S_2')$, $\Theta(S_2, S_2')$ and $\Theta(S_3, S_2')$, but only $M + N = 5$ rules.

Without using complex states, a "naive" parallel composed system, $\Pi_1 \times \Pi_2$, needs $M \cdot N$ states and $O(M \cdot N)$ rules, shown as follows.

- $\Pi_1 \times \Pi_2$, also has 6 $(= 3 \cdot 2)$ states but 18 $(= 3 \cdot 2 \cdot 3)$ rules:

$$
\begin{array}{lll}
S_{11}\, a\, d & \rightarrow_{\texttt{min}} & S_{22}\, b\, e \\
S_{11}\, a & \rightarrow_{\texttt{min}} & S_{21}\, b \\
S_{11}\, d & \rightarrow_{\texttt{min}} & S_{12}\, e \\
S_{12}\, a\, e & \rightarrow_{\texttt{min}} & S_{21}\, b\, d \\
S_{12}\, a & \rightarrow_{\texttt{min}} & S_{22}\, b \\
S_{12}\, e & \rightarrow_{\texttt{min}} & S_{11}\, d \\
S_{21}\, b\, d & \rightarrow_{\texttt{min}} & S_{32}\, c\, e \\
S_{21}\, b & \rightarrow_{\texttt{min}} & S_{31}\, c \\
S_{21}\, d & \rightarrow_{\texttt{min}} & S_{22}\, e
\end{array}
\qquad
\begin{array}{lll}
S_{22}\, b\, e & \rightarrow_{\texttt{min}} & S_{31}\, c\, d \\
S_{22}\, b & \rightarrow_{\texttt{min}} & S_{32}\, c \\
S_{22}\, e & \rightarrow_{\texttt{min}} & S_{21}\, d \\
S_{31}\, c\, d & \rightarrow_{\texttt{min}} & S_{12}\, a\, e \\
S_{31}\, c & \rightarrow_{\texttt{min}} & S_{11}\, a \\
S_{31}\, d & \rightarrow_{\texttt{min}} & S_{32}\, e \\
S_{32}\, c\, e & \rightarrow_{\texttt{min}} & S_{11}\, a\, d \\
S_{32}\, c & \rightarrow_{\texttt{min}} & S_{12}\, a \\
S_{32}\, e & \rightarrow_{\texttt{min}} & S_{31}\, d
\end{array}
$$

Note that, although $\Pi_1 \parallel \Pi_2$ has in general an order of magnitude fewer rules than $\Pi_1 \times \Pi_2$, as $O(M + N) \ll O(M \cdot N)$, their state sets are isomorphic. Figure 2.5 shows state charts for Π_1, Π_2 and $\Pi_1 \times \Pi_2 \overset{states}{\simeq} \Pi_1 \parallel \Pi_2$.



(a) Π_1 (b) Π_2 (c) $\Pi_1 \times \Pi_2 \overset{states}{\simeq} \Pi_1 \parallel \Pi_2$

Figure 2.5: State charts of Π_1, Π_2 and $\Pi_1 \times \Pi_2 \overset{states}{\simeq} \Pi_1 \parallel \Pi_2$.

Parallel Composition with Interaction

Parallel composition with interaction of two systems, Π_1 and Π_2, can be considered as running in parallel Π_1 and Π_2, where Π_1 "feeds" symbols to Π_2. This parallel composition is essential to cleanly add a separate *control layer*, Π_2, over any *arbitrary* algorithm, Π_1. In this thesis, this composition is specifically used in Section 3.6, to augment an implicitly terminating algorithm (Π_1) with a termination detection algorithm (Π_2), to obtain a partial-explicitly terminating algorithm (§ 1.2).

Consider Π_1 and Π_2, which satisfy the following conditions:

- Π_1 and Π_2 use disjoint sets of states (if not, without loss of generality, we can relabel the states to satisfy this condition);
- Π_1 and Π_2 share a set of symbols on three conditions:
  - initially, no left-side symbols or promoter symbols of Π_2 are available;
  - no rule of Π_2 has empty left-side symbols;
  - no rule of Π_2 generates symbols of Π_1 or symbols generated by Π_2 do not affect Π_1.

The parallel composition of Π_1 and Π_2 with interaction, denoted as $\Pi_1 \triangleright \Pi_2$, is constructed in the same way as $\Pi_1 \parallel \Pi_2$. In this composition, it is important to ensure that rules of Π_1 have higher priority than rules of Π_2, unlike in the parallel composition with no interaction, where the priority does not matter.

Example 2.14 illustrates an example. Π_1 cycles over three states and each iteration generates one symbol, b, and Π_2 cycles over two states and each iteration transforms one b to one d. In $\Pi_1 \triangleright \Pi_2$, Π_2 transforms symbol b generated by Π_1 in each iteration to symbol d, therefore obtaining one d in each iteration.

Example 2.14.

- Π_1, has 3 states and 3 rules:

$$\begin{array}{lll} S_1\, a & \rightarrow_{\texttt{min}} & S_2\, b\, e \\ S_2\, e & \rightarrow_{\texttt{min}} & S_3\, f \\ S_3\, f & \rightarrow_{\texttt{min}} & S_1\, a \end{array}$$

  Step-by-step evolution:

  $S_1\, a \Rightarrow S_2\, b\, e \Rightarrow S_3\, b\, f \Rightarrow S_1\, b\, a \Rightarrow S_2\, b^2\, e \Rightarrow \cdots \Rightarrow S_1\, b^n\, a$ (after n iterations)

- Π_2, has 2 states and 2 rules:

$$\begin{array}{lll} S'_1\, b & \rightarrow_{\texttt{min}} & S'_2\, c \\ S'_2\, c & \rightarrow_{\texttt{min}} & S'_1\, d \end{array}$$

  Step-by-step evolution:

  $S'_1\, b \Rightarrow S'_2\, c \Rightarrow S'_1\, d$

- $\Pi_1 \triangleright \Pi_2$, has 4 $(= 2 + 2)$ states and 5 $(= 3 + 2)$ rules:

$$\begin{array}{lll} \overline{\Pi_1}: & & \\ r_1 : \Theta(S_1, Y)\, a & \rightarrow_{\texttt{min}} & \Theta(S_2, Y)\, b\, e \\ r_2 : \Theta(S_2, Y)\, e & \rightarrow_{\texttt{min}} & \Theta(S_3, Y)\, f \\ r_3 : \Theta(S_3, Y)\, f & \rightarrow_{\texttt{min}} & \Theta(S_1, Y)\, a \end{array}$$

$$\begin{array}{lll} \overline{\Pi_2}: & & \\ r_4 : \Theta(X, S'_1)\, b & \rightarrow_{\texttt{min}} & \Theta(X, S'_2)\, c \\ r_5 : \Theta(X, S'_2)\, c & \rightarrow_{\texttt{min}} & \Theta(X, S'_1)\, d \end{array}$$

  Step-by-step evolution:

  $\Theta(S_1, S'_1)\, a \Rightarrow \Theta(S_2, S'_2)\, e\, c \Rightarrow \Theta(S_3, S'_1)\, f\, d \Rightarrow \Theta(S_1, S'_1)\, a\, d \Rightarrow \Theta(S_2, S'_2)\, b\, e\, d \Rightarrow \cdots \Rightarrow \Theta(S_1, S'_1)\, a\, d^n$ (after n iterations)

  Now we explain the first step evolution, $\Theta(S_1, S'_1)\, a \Rightarrow \Theta(S_2, S'_2)\, e\, c$, and the mapping of target state by weak binding. Consider the above rules of $\Pi_1 \triangleright \Pi_2$ in a synchronous system where cell σ_1 in state $\Theta(S_1, S'_1)$ contains one symbol a.

  1. First, rule r_1 is applied: one a is consumed and be become immediately available; σ_1's target state is temporarily mapped to $\Theta(S_2, Y)$.
  2. Next, rule r_4 is applied: one b is consumed and one c becomes immediately available; σ_1's target state is now fixed to $\Theta(S_2, S'_2)$.
  3. At the end of the step, σ_1 is in state $\Theta(S_2, S'_2)$ and contains multiset ec.

$\Pi_1 \triangleright \Pi_2$ composition generates 6 states: $\Theta(S_1, S_1')$, $\Theta(S_2, S_1')$, $\Theta(S_3, S_1')$, $\Theta(S_1, S_2')$, $\Theta(S_2, S_2')$ and $\Theta(S_3, S_2')$. However, some are not reachable due to the sharing of symbols; only three states are used: $\Theta(S_1, S_1')$, $\Theta(S_2, S_2')$ and $\Theta(S_3, S_1')$.

Example 2.15. The following are properties of sequential and parallel compositions; other associative and commutative properties remain an open question.

- $(\Pi_1; \Pi_2); \Pi_3 = \Pi_1; (\Pi_2; \Pi_3)$,
- $(\Pi_1 \parallel \Pi_2) \parallel \Pi_3 = \Pi_1 \parallel (\Pi_2 \parallel \Pi_3)$,
- $(\Pi_1 \parallel \Pi_2) \triangleright \Pi_3$,
- $(\Pi_1; \Pi_2) \triangleright \Pi_3$,
- $\Pi_1; (\Pi_2 \triangleright \Pi_3)$.
- $\Pi_1; (\Pi_2 \parallel \Pi_3)$.

2.4 Model Comparisons

We have previously discussed existing P systems framework for modelling distributed algorithms (§1.3); in this section, we continue to compare other computational models with our xP systems.

IO automata: IO automata are used to describe most types of asynchronous concurrent systems [42]. The specification of IO automata consists of a set of states, actions and transitions. Action can be input, internal or output actions, which are consistent with the Receive, Process and Send substeps in P systems. Due to the lack of readability of their formal specification, IO automata are usually written using precondition-effect notation, in a semi-pseudocode style that is non-executable. This notation style groups a set of transitions into one piece of code, which is executed indivisibly and similar to the atomic step of xP systems.

Figure 2.6 compares an xP system with its equivalent IO automata for AsynchBFS algorithm (§ 3.4.3). Even thought at the executable level, our formal xP system uses only 14 rules while the informal IO automata specification uses 27 lines.

Cellular automata: In cellular automata, there is a fixed bound (indegree and outdegree) on the number of input and output connections of each cell and each step uses one single finite automata rule; in P systems, each step works on unbounded multiset and uses a whole ruleset, which seems more powerful. In cellular automata, inputs are organised as arrays, not multisets as in P systems, so a cell knows exactly from which neighbour the input comes from, which is difficult to replicate in P systems. Previous work [33] on firing squad problems shows that P systems solution is crisper and more realistic; however, specifically for directed graphs, the cellular automata solution is very large, using long message trains.

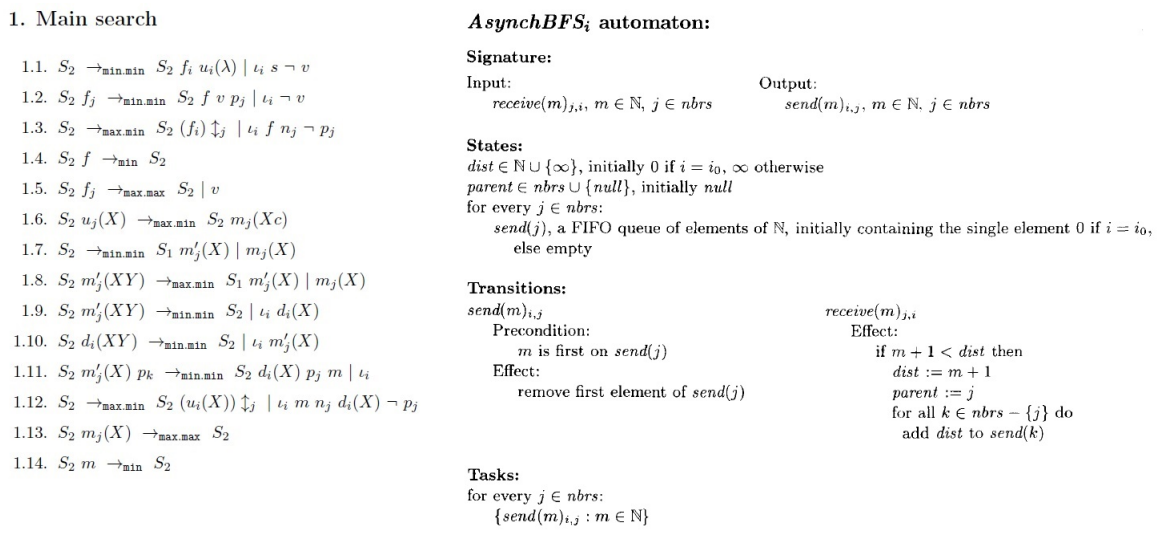


Figure 2.6: An xP system and its equivalent IO automata for AsynchBFS algorithm.

Petri nets: Petri nets consist of *places* and *transitions*. Places indicate the local availability of resources, which can be used to represent symbols in specific compartments in P systems; transitions are actions which can occur depending on local conditions related to the availability of resources, which can be used to represent rewriting rules associated with specific cells P systems [37]. However, because a separate place, (x, j), is needed for each symbol, x, and cell σ_j, the number of places and communication channels to present a P system can be very large. Recent research has investigated the equality and conversion of P systems and Petri nets: the maximally concurrent computation of P systems can be reflected by the maximal concurrency semantics of a class of Petri nets with localities [37]; the computational completeness of P systems with priorities and zero-test using symbol objects can be studied through Petri nets [27]; spiking neural (SN) P systems with various features can be represented by Petri nets [46, 45].

2.5 Summary

This chapter introduces xP systems, which are extended versions of simple P systems. The features of xP systems are summarised below.

- The membrane structure is a *digraph*.
- Structural *parent* $\rightarrow$ *child* arcs define *duplex* communication channels.
- All cells share the *same* symbol set, state set and ruleset.
- Each cell is "blessed" with a unique cell ID symbol.

- The rule applications of a vector is performed in one step.

Our xP systems provide useful ingredients for modelling distributed algorithms:

- Our unified formal model of both synchronous and asynchronous systems uses the same static description for both synchronous and asynchronous systems and only the messaging delays differ, which matches the definitions in distributed computing [62, 42].
- Our proposal λ messaging makes xP systems conform to activation assumptions in distributed computing (§ 1.2).
- Complex symbols and cell IDs allow complex data structures required by complex algorithms.
- Immediately available"here" symbols, combined *instantiation.rewriting* mode of generic rules and matrix structured rulesets provide ingredients to achieve zero local computation time, which is one of the assumptions of common time complexity (§ 1.2), unlike traditional P systems, in which the required local computation may take a number of time units.
- Complex states and rule fragments composition allow SoC designs for complex distributed problems that can be divided into smaller problems.
- Complex symbols and cell IDs enable fixed-size alphabets and rulesets.

In the following chapters, xP systems are used to model and improve large practical applications, ranging from fundamental graph traversal problems [3] and distributed termination detection problems (Chapter 3) to complex graph theoretical problems [51, 52, 22, 65] (Chapter 4) and to one of the most challenging distributed algorithms, the minimum spanning tree problem [64] (Chapter 5).

Chapter 3

Traversal Algorithms

This chapter first presents our xP solutions by reformulating the simple P system solutions of our joint work by Bălănescu, Nicolescu and Wu [3] for several fundamental distributed traversal algorithms: (1) Echo and a set of distributed depth-first search (DFS), which work in both synchronous and asynchronous settings; (2) distributed synchronous breadth-first search (SynchBFS) and asynchronous breadth-first search (AsynchBFS) algorithms. All of these traversal algorithms assume that each cell knows all its neighbours. Next, we provide synchronous neighbour discovery (SynchND) and asynchronous neighbour discovery (AsynchND) algorithms and their xP solutions, based on our joint work [3]. Moreover, we address a common problem in distributed algorithms, the termination detection problem. As new contributions in this thesis, we present xP solutions of several well-known termination detection algorithms and their applications; by using rule fragments composition (§ 2.3.7), we obtain complete solutions of traversal algorithms, which include both neighbour discovery and termination detection.

This chapter is organised in the following way. Following our joint work [3], Section 3.2 provides an xP solution of the Echo algorithm, which was introduced in Section 1.1; Section 3.3 discusses distributed DFS algorithms and their xP solutions; Section 3.4 discusses SynchBFS and AsynchBFS algorithms and their xP solutions; Section 3.5 presents SynchND and AsynchND algorithms and their xP solutions, in which no cell knows the algorithm termination. As my new contributions in this thesis, Section 3.6 describes several termination detection algorithms and presents their xP solutions and applications for algorithms SynchBFS, AsynchBFS and AsynchND; Section 3.7 gives complete xP solutions of traversal algorithms, which include both neighbour discovery and termination detection. Finally, Section 3.8 summarises this chapter and compares our xP specifications with pseudocodes of the corresponding distributed algorithms presented in Tel's book [62], in terms of runtime complexity and program size.

3.1 Graph in xP Systems

In this thesis, all graphs are *connected* and all digraphs are *weakly connected*. All our xP solutions of distributed graph algorithms are *fully distributed*.

1. The graph or digraph is represented as the underlying structure of an xP system (§ 2.2).
2. There is no central cell to convey global information among cells, i.e. cells only communicate with neighbours via local channels (between structural neighbours).
3. Neighbours, tree parents and children or paths predecessors and successors are recorded locally in each cell, similar to distributed routing tables in networks.

In this chapter, all traversal algorithms work on *undirected* graphs. Initially, we assume that each cell already knows all its neighbours; we later augment these algorithms with a preliminary phase which builds this knowledge.

All our xP specifications of traversal algorithms share a *uniform design* based on identical or similar symbols and rulesets. The neighbourhood information and spanning tree information are *reified* as pointer symbols:

- n_j—indicating that σ_j is a neighbour;
- p_k—indicating that σ_k is a spanning tree parent.

Table 3.1 shows distributed "routing" records for one possible spanning tree of Figure 1.3 (i). Specifically, the root of the spanning tree, cell σ_i, has a parent pointer to itself, p_i. Other graph information can be represented in a similar way.

Table 3.1: Distributed "routing" records for one possible spanning tree of the Echo algorithm for Figure 1.3 (i).

Cell	σ_1	σ_2	σ_3	σ_4
ID	ι_1	ι_2	ι_3	ι_4
Neighbours	$n_2\ n_3\ n_4$	$n_1\ n_3\ n_4$	$n_1\ n_2\ n_4$	$n_1\ n_2\ n_3$
Tree parent	p_1	p_1	p_2	p_3

3.2 The Echo Algorithm

The Echo algorithm works in both synchronous and asynchronous modes, as we briefly described in Section 1.1. Here we explain this algorithm in detail below, where (I) performs the forward phase and (II) performs the return phase (§ 1.1).

Each cell keeps a *counter* for its expected visit tokens, initially set to zero. The source cell starts by broadcasting visit tokens to all its neighbours. An *unvisited* cell receiving a visit token becomes a *frontier* cell.

(I) When an *unvisited* cell receives visit tokens, it selects *one* of the sending cells as its parent and marks itself as visited, becoming a *frontier* cell. Then:

  (Ia) it sends visit tokens to all non-parent neighbours; for each sent visit token, it increments its counter by one.

(II) When a *visited* cell receives a visit token, it decrements its counter by one; if its counter reaches zero, i.e. it receives all expected visit tokens, it sends a visit token to its parent.

When the source cell receives all expected visit tokens, the algorithm terminates. In this algorithm, the source cell knows the algorithm termination, but all non-source cells are not aware of the algorithm termination.

In a parallel run, an unvisited cell, σ_i, may receive several visit tokens *simultaneously*, but only *one* of sending cells can successfully become σ_i's parent. This *race condition* is a common problem in distributed algorithms: such as SynchBFS and AsynchBFS in Section 3.4, BFS-based disjoint paths algorithms in Sections 4.5 and 4.6 and the Dijkstra-Scholten algorithm in Section 3.6.3. A solution is using a `min.min` mode; such a `min.min` rule is instantiated only once, for an arbitrary choice among all received messages [52] (see rule 1.2 of xP Specification 3.1 for an example).

Example 3.1. Figure 3.1 shows such a race condition: cell σ_4 receives visit tokens from cells σ_1, σ_2 and σ_3 simultaneously; but only one of the sending cells, e.g., cell σ_1, becomes σ_4's parent.

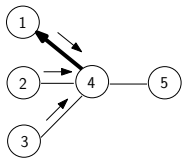


Figure 3.1: A race condition when an unvisited cell, σ_4, receives several visit tokens simultaneously. Thin edges: graph edges; thick arc: virtual spanning tree child-parent arc; arrows beside edges: visit tokens received by σ_4.

xP Specification 3.1: Echo

Input: All cells start in the same initial state, S_2, with the same set of rules. Each cell, σ_i, contains an immutable cell ID symbol, ι_i, and neighbour pointers, n_j's. The source cell, σ_s, is additionally marked with one symbol, s.

Output: All cells end in the same state, S_3; neighbour pointer symbols and cell IDs are intact. Cell σ_s is still marked with one s. Each cell contains a spanning tree parent pointer, p_j. Specifically, the source cell, σ_s, contains one p_s indicating it is the root of the spanning tree and one g indicating it knows the algorithm termination.

Symbols and states

Cell σ_i uses the following symbols:

- n_j indicates its neighbour, σ_j;
- p_j indicates its spanning tree parent, σ_j;
- f_i is its visit token;
- f indicates that it is a frontier cell;
- w is used for its counter to wait for expected visit tokens.

State S_2 is an unvisited state and S_3 is a visited state.

The matrix R of xP Specification 3.1 consists of two vectors, informally presented in two groups, according to their functionality and applicability. Each vector implements an independent function, performed in one step.

1. An unvisited cell receives tokens.

1.1. $S_2 \ \rightarrow_{\texttt{min.min}} \ S_3 \ f_i \mid \iota_i \ s \ \neg \ p_i$

1.2. $S_2 \ f_j \ \rightarrow_{\texttt{min.min}} \ S_3 \ p_j \ f$

1.3. $S_2 \ \rightarrow_{\texttt{max.min}} \ S_3 \ w \ (f_i) \updownarrow_j \ \mid \iota_i \ f \ n_j \ \neg \ p_j$

1.4. $S_2 \ f \ \rightarrow_{\texttt{min}} \ S_3$

2. A visited cell receives tokens.

2.1. $S_3 \ w \ \rightarrow_{\texttt{max.min}} \ S_3 \mid f_k$

2.2. $S_3 \ \rightarrow_{\texttt{min.min}} \ S_3 \ (f_i) \updownarrow_j \ \mid \iota_i \ f_k \ p_j \ \neg \ w$

2.3. $S_3 \ f_k \ \rightarrow_{\texttt{max.max}} \ S_3$

2.4. $S_3 \ \rightarrow_{\texttt{min}} \ S_3 \ g \mid s \ \neg \ w$

Initial and final configurations

Table 3.2 shows the initial and final configurations of xP Specification 3.1 for Figure 1.2 (synchronous mode) and Figure 1.3 (asynchronous mode), highlighting the spanning trees.

Table 3.2: Initial and final configurations of xP Specification 3.1 for Figure 1.2 (synchronous mode) and Figure 1.3 (asynchronous mode).

Time	σ_1	σ_2	σ_3	σ_4
0	$S_2\ s\ \iota_1\ n_2\ n_3\ n_4$	$S_2\ \iota_2\ n_1\ n_3\ n_4$	$S_2\ \iota_3\ n_1\ n_2\ n_4$	$S_2\ \iota_4\ n_1\ n_2\ n_3$
Synch. 3	$S_3\ s\ \iota_1\ n_2\ n_3\ n_4\ \mathbf{p_1}\ g$	$S_3\ \iota_2\ n_1\ n_3\ n_4\ \mathbf{p_1}$	$S_3\ \iota_3\ n_1\ n_2\ n_4\ \mathbf{p_1}$	$S_3\ \iota_4\ n_1\ n_2\ n_3\ \mathbf{p_1}$
Asynch. 4	$S_3\ s\ \iota_1\ n_2\ n_3\ n_4\ \mathbf{p_1}\ g$	$S_3\ \iota_2\ n_1\ n_3\ n_4\ \mathbf{p_1}$	$S_3\ \iota_3\ n_1\ n_2\ n_4\ \mathbf{p_2}$	$S_3\ \iota_4\ n_1\ n_2\ n_3\ \mathbf{p_3}$

Rule explanations

The xP rules correspond to the previous algorithm descriptions. The source cell, σ_s, generates one f_s, which simulates that σ_s receives a visit token from a non-existing cell (rule 1.1).

(I) Rule 1.2: when unvisited σ_i (in state S_2) receives f_j's, it selects one of the sending cells, σ_j, by using `min.min` mode, as its parent, p_j, marks itself as visited by entering state S_3, and generates one f, indicating it is a frontier cell.

(Ia) Rules 1.3–4: σ_i sends f_i to all non-parent neighbours, $n_j \neg p_j$, and generates one w for each sent f_i (rule 1.3); then σ_i erases f (rule 1.4).

(II) Rules 2.1–2.3: when visited σ_i (in state S_3) receives f_k's, it erases one w for each received f_k (rule 2.1); if σ_i receives all expected visit tokens, $\neg\ w$, it sends f_i to its parent, p_j (rules 2.2–3).

When the source cell, σ_s, receives all its expected tokens, $\neg\ w$, it generates one g, indicating it knows the algorithm termination (rule 2.4).

Partial traces

Table 3.3 shows the partial traces of of xP Specification 3.1 for cell σ_3 in Figure 1.3 (asynchronous mode), where the "Evolution" column shows the *cell content evolution* at each step (discussed below) and the "Content" column shows the cell content after the evolution of the step.

To explain the cell content evolution at a step, a form $\{r\}\ c \Rightarrow g\ \{l\}\ \{m\}_R \ldots$ is used, where

- received message r is consumed;
- multiset c is consumed;
- multiset g becomes immediately available in the same cell;
- message l is a loopback message sent to the same cell;

- message m is sent to neighbour cells indicated by R;
- ellipses $(\dots)$ indicate possible repetitions of the last parenthesised item.

Note that this form of evolution is different from the rule form: it is the cell content evolution at one step that includes one or more rule applications, omitting intermediate symbols that are generated and later consumed during the same step.

Table 3.3: Partial traces of xP Specification 3.1 for cell σ_3 in Figure 1.3 (one possible asynchronous run). Omitted symbols $(\dots)$ are $\iota_3\ n_1\ n_2\ n_4$.

Fig.	Evolution	Content
(a) (b)		$\dots$
(c)	$\{f_2\} \Rightarrow p_2\ w^2\ \{f_3\}_{1,4}$	$p_2\ w^2 \dots$
(d) (e)		$p_2\ w^2 \dots$
(f)	$\{f_1\}\ w \Rightarrow$	$p_2\ w \dots$
(g)	$\{f_4\}\ w \Rightarrow \{f_3\}_2$	$p_2 \dots$
(h) (i)		$p_2 \dots$

Take row 2, Fig.(c), in Table 3.3 for example. At this step, when unvisited cell σ_3 receives a visit token, f_2, from σ_2, it (i) sets its parent as p_2, becoming a frontier by generating one f (rule 1.2); (ii) sends visit token f_3 to σ_1 and σ_4, incrementing its counter by two w's (rule 1.3); and (iii) erases f (rule 1.4). This cell content evolution is denoted by $\{f_2\} \Rightarrow p_2\ w^2\ \{f_3\}_{1,4}$. After this evolution, σ_3's content contains $p_2\ w^2$. Note that the intermediate symbol, f, is not shown by this form.

Moreover, for the step where there is no cell content evolution, if no message is received, a cell's content remains the same as the previous step, e.g., row 3, Fig. (d) (e); otherwise, a cell's content is augmented with received messages.

Smaller program size with complex symbols

By using complex symbols, our xP specification has 8 rules. In contrast, using elementary symbols, the synchronous P specification proposed by Kim [35] uses 28 rules, which does not require any topology knowledge in each cell. As a fair comparison, an xP solution augmented with a preliminary neighbour discovery phase, SynchND (§ 3.5), uses only $8+3=11$ rules in total, showing substantially smaller program size than the P solution using elementary symbols.

3.3 Distributed Depth-First Search Algorithms

Depth-first search (DFS) and breadth-first search (BFS) are graph traversal algorithms, which construct a DFS spanning tree and a BFS spanning tree, respectively. Figure 3.2 shows one possible *virtual* DFS spanning tree, T, in an undirected graph, G.

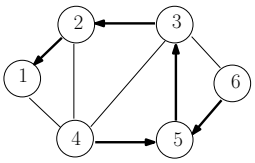


Figure 3.2: One possible virtual DFS spanning tree in an undirected graph. Thick arrows indicate virtual spanning tree child-parent arcs.

DFS is a fundamental technique, apparently inherently *sequential*. Several distributed DFS algorithms have been proposed, which attempt to make DFS run faster on distributed systems, such as the classical DFS [62], Awerbuch's DFS [2], Cidon's DFS [13], Sharma et al.'s DFS [61] and Makki et al.'s DFS [43].

Various P solutions for the classical DFS have been proposed. Dinneen et al. proposed a P solution to find disjoint paths in a digraph, using the classical DFS, which marks a cell as visited by changing its state [18]. Bernardini et al. proposed a P synchronization solution based on the classical DFS [4], where a cell marks itself as visited by changing the polarity from 0 to + and implicitly records which neighbour cells have been visited or not. Gutiérrez-Naranjo et al.'s P solution for the classical DFS [31] uses inhibitors to avoid visiting already-visited neighbour cells; however, it uses graph encoded symbols instead of P system structure and thus is not a distributed solution.

Following our joint work [3], this section provides xP solutions of several distributed DFS algorithms [3], which represent graphs by the underlying structures of xP systems and mark a cell as visited by a dedicated symbol.

3.3.1 DFS Token Handling Rules in Undirected Graphs

DFS explores as far as possible along each branch before backtracking, by one *single token* that plays two different roles: *forward* and *backtrack*. We call the token according to its role:

- *forward token*: the token in the forward role;
- *backtrack token*: the token in the backtrack role.

DFS starts by the source cell's sending a forward token to one of its children. A cell handles a received forward or backtrack token based on following *token handling rules*.

(I) When an *unvisited* cell, σ_i, receives a *forward* token from its neighbour cell, σ_j, it sets its *parent* as σ_j and marks itself as visited, becoming a *frontier* cell.

(II) When a *visited* cell, σ_i, receives a *backtrack* token from one of its *children*, σ_j, it becomes a *frontier* cell.

(III) When *visited* cell, σ_i, receives a *forward* token from its neighbour cell, σ_j, it handles the token according to the specific algorithm, e.g., in the classical DFS, it sends its backtrack token to σ_j or in Cidon's DFS, it just ignores the token.

If cell σ_i becomes a frontier cell after (I) or (II), then it sends its token as in (a) or (b).

(a) If σ_i has any unvisited non-parent neighbour, σ_k, it sends its *forward* token to σ_k.

(b) Otherwise, it sends its *backtrack* token to its parent.

Except the classical DFS, all other distributed DFS algorithms attempt to avoid case (III), i.e. sending a forward token to a visited cell. As later discussed, except the classical DFS and Cidon's DFS, where case (III) may still appear, all other algorithms, Awerbuch's DFS, Sharma et al.'s DFS and Makki et al.'s DFS, completely avoid this case.

Specifically for (b), Makki et al.'s DFS uses an evolved version to speed up the backtrack process, which may not necessarily backtrack to a cell's parent (as later discussed in Section 3.3.6).

Example 3.2. Figure 3.3 shows how a cell handles a received forward or backtrack token, based on the token handling rules, where cell σ_2 becomes a frontier cell in scenarios (Ia), (Ib), (IIa) and (IIb).

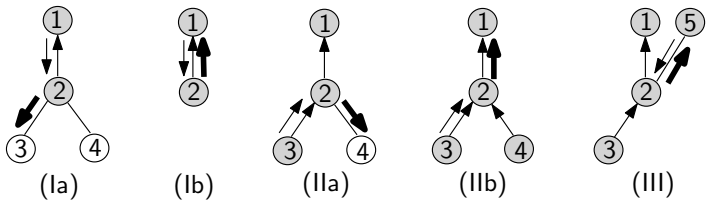


Figure 3.3: A cell, σ_2, handles its received forward or backtrack token in distributed DFS. Edges with arrows: virtual spanning tree child-parent arcs; gray cells: visited cells; thin arrows beside edges: tokens received by σ_2; thick arrows beside edges: tokens sent by σ_2.

(Ia) When an unvisited cell, σ_2, receives a forward token from σ_1, it sets its parent as σ_1 and marks itself as visited, becoming a frontier cell. Cell σ_2 has unvisited non-parent neighbours, so it sends its forward token to one of them, e.g., σ_3.

(Ib) When an unvisited cell, σ_2, receives a forward token from σ_1, it sets its parent as σ_1 and marks itself as visited, becoming a frontier cell. Cell σ_2 has no unvisited non-parent neighbours, so it sends its backtrack token to its parent, σ_1.

(IIa) When a visited cell, σ_2, receives a backtrack token from its child, σ_3, it becomes a frontier cell. Cell σ_2 has an unvisited non-parent neighbour, σ_4, so it sends its forward token to σ_4.

(IIb) When a visited cell, σ_2, receives a backtrack token from one of its children, σ_3, it becomes a frontier cell. Cell σ_2 has no unvisited non-parent neighbours, so it sends its backtrack token to its parent, σ_1.

(III) When a visited cell, σ_2, receives a forward token from σ_5, it handles the token according to the specific algorithm, e.g., it sends its backtrack token to σ_5, as in the classical DFS.

All DFS algorithms terminate when the source cell receives a backtrack token and it has no more unvisited neighbours. The source cell knows when the algorithm terminates, but all non-source cells do not know the algorithm termination. If required, the algorithm can be supplemented by a broadcast phase, which announces the algorithm termination.

3.3.2 Classical DFS

The classical DFS algorithm [62] is based on Tarry's traversal algorithm [62], which traverses all arcs *sequentially* in both directions using one single token.

Each cell records a list of visited neighbours, to which it has sent a token. Thus, the unvisited neighbours, which are not in its recorded visited list, are only *possibly-unvisited* (they may have already been visited by other cells). Because the classical DFS traverses each arc twice, serially, it is not the most efficient distributed DFS algorithm [3].

The classical DFS algorithm works in both asynchronous and synchronous modes; for simplicity, we explain it in a synchronous scenario.

Example 3.3. Figure 3.4 shows one snippet of the evolution of the classical DFS algorithm in the synchronous mode. Assume that the search started from cell σ_1 and just backtracked to cell σ_5, after building search path $\sigma_1.\sigma_2.\sigma_3.\sigma_5.\sigma_6$.

(a) Cell σ_5 sends its forward token to the only one remaining possibly-unvisited cell, σ_4, and marks σ_4 as visited.

(b) On receiving the forward token from σ_5, unvisited cell σ_4 sets its parent as σ_5, marks itself as visited, sends its forward token to one of its possibly-unvisited neighbours, σ_1, and marks σ_1 as visited.

(c) On receiving the forward token from σ_4, because σ_1 has already been visited, it sends back a backtrack token to σ_4 and marks σ_4 as visited.

(d) On receiving the backtrack token from σ_1, cell σ_4 sends its forward token to one of its possibly-unvisited neighbours, σ_2, and marks σ_2 as visited.

(e) On receiving the forward token from σ_4, because σ_2 has already been visited, it sends back a backtrack token to σ_4 and marks σ_4 as visited.

(f) On receiving the backtrack token from σ_2, cell σ_4 sends its forward token to one of its possibly-unvisited neighbours, σ_3, and marks σ_3 as visited.

(g) On receiving the forward token from σ_4, because σ_3 has already been visited, it sends back a backtrack token to σ_4 and marks σ_4 as visited.

(h) On receiving the backtrack token from σ_3, cell σ_4 has no more possibly-unvisited non-parent neighbours, so it sends a backtrack token to its parent, σ_5, and marks σ_5 as visited.

Note that, in this algorithm, cell σ_4 needlessly probes cells σ_1, σ_2 and σ_3, which have already been visited.

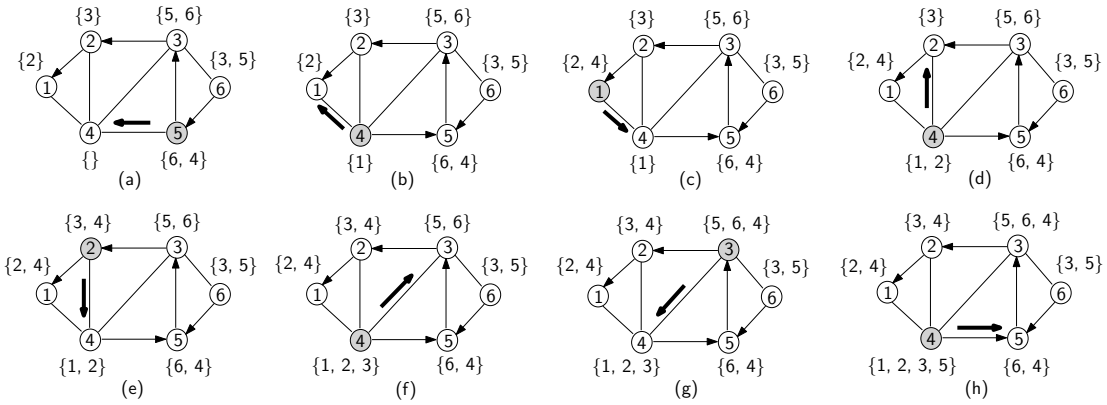


Figure 3.4: An example of the classical DFS in the synchronous mode. Arcs with arrows: virtual spanning tree child-parent arcs; thick arrows beside arcs: tokens; gray cells: frontier cells; elements in braces beside each cell: neighbours recorded as visited.

xP Specification 3.2: Classical DFS

Input: Same as in xP Specification 3.1.

Output: All cells end in the same state, S_3; neighbour pointer symbols and cell IDs are intact. Cell σ_s is still marked with one s. Each cell contains a DFS spanning tree parent pointer, p_j. Specifically, the source cell, σ_s, contains one p_s indicating it is the root of the spanning tree and one g indicating it knows the algorithm termination.

Symbols and states

Our xP Specification 3.2 of the classical DFS uses the following symbols and states, which are also used in the same way in xP Specifications 3.3, 3.4, 3.5 and 3.6:

- f_i is its token;
- n_j indicates its neighbour, σ_j;
- p_j indicates its spanning tree parent, σ_j;
- f indicates that it is a frontier cell;
- v indicates that it is visited;
- g indicates that it must next clean up useless symbols; specifically, g in the source cell, σ_s, indicates that σ_s knows termination.

State S_2 is used as both unvisited and visited states; S_3 is a state after cleaning up all useless symbols.

Specifically in the classical DFS, cell σ_i uses v_j to record the neighbour to which σ_i has sent its token.

Consider a cell, σ_i, which receives a token, f_j: (1) a forward token is indicated by $f_j \neg v_j$, and (2) a backtrack token is indicated by $f_j\ v_j$ (σ_i must have sent a token to σ_j before).

The matrix R of xP Specification 3.2 consists of two vectors, informally presented in two groups, according to their functionality and applicability. Each vector implements an independent function, performed in one step. The same matrix structure (consisting of one vector for main search and one vector for clean-up) is used in all other distributed DFS xP specifications in this chapter.

1. Main search

1.1. $S_2 \rightarrow_{\texttt{min.min}} S_2\ f_i \mid \iota_i\ s \neg v$

1.2. $S_2\ f_j \rightarrow_{\texttt{min.min}} S_2\ f\ v\ p_j \neg v\ v_j$

1.3. $S_2\ f_j \rightarrow_{\texttt{min.min}} S_2\ f \mid v\ v_j$

1.4. $S_2\ f_j \rightarrow_{\texttt{min.min}} S_2\ v_j\ (f_i) \updownarrow_j \mid \iota_i\ v \neg v_j$

1.5. $S_2\ f \rightarrow_{\texttt{min.min}} S_2\ v_j\ (f_i) \updownarrow_j \mid \iota_i\ n_j \neg p_j\ v_j$

1.6. $S_2\ f \rightarrow_{\texttt{min.min}} S_2\ g\ v_j\ (f_i) \updownarrow_j \mid \iota_i\ p_j \neg s$

1.7. $S_2\ f \rightarrow_{\texttt{min}} S_2\ g \mid s$

2. Clean-up

2.1. $S_2\ v_j\ \rightarrow_{\texttt{max.max}}\ S_3 \mid g$

2.2. $S_2\ v\ \rightarrow_{\texttt{max}}\ S_3 \mid g$

2.3. $S_2\ g\ \rightarrow_{\texttt{min}}\ S_3 \mid \neg\ s$

Initial and final configurations

Table 3.4 shows the initial and final configurations of xP Specification 3.2 for Figure 3.4, highlighting one possible DFS spanning tree. Omitted symbols (...) in the final configuration are the neighbour pointers, n_j's, which are the same as in the initial configuration.

Table 3.4: Initial and final configurations of xP Specification 3.2 for Figure 3.2.

Time	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_2\ s\ \iota_1 n_2 n_4$	$S_2\ \iota_2 n_1 n_3 n_4$	$S_2\ \iota_3 n_2 n_4 n_5 n_6$	$S_2\ \iota_4 n_1 n_2 n_3 n_5$	$S_2\ \iota_5 n_3 n_4 n_6$	$S_2\ \iota_6 n_3 n_5$
19	$S_3\ s\ \iota_1 \boldsymbol{p_1}\ g \ldots$	$S_3\ \iota_2 \boldsymbol{p_1} \ldots$	$S_3\ \iota_3 \boldsymbol{p_2} \ldots$	$S_3\ \iota_4 \boldsymbol{p_5} \ldots$	$S_3\ \iota_5 \boldsymbol{p_3} \ldots$	$S_3\ \iota_6 \boldsymbol{p_5} \ldots$

Rule explanations

The source cell, σ_s, generates one token, f_s, which simulates that σ_s receives a forward token from a non-existing cell (rule 1.1). Cell σ_i handles the received token, f_j, using the following rules corresponding to the token handling rules discussed in Section 3.3.1.

(I) Rule 1.2: when unvisited σ_i, $\neg\ v$, receives a forward token, $f_j\ \neg\ v_j$, it sets its parent as p_j, marks itself as visited by v and generates one f, indicating it is a frontier cell.

(II) Rule 1.3: when visited σ_i, v, receives a backtrack token, $f_j\ v_j$, it generates one f.

(III) Rule 1.4: when visited σ_i, v, receives a forward token, $f_j\ \neg\ v_j$, it sends back f_i to σ_j and marks σ_j as visited by v_j.

As a frontier, σ_i sends its token as follows.

(a) Rule 1.5: if σ_i has any unvisited non-parent neighbour, $n_j\ \neg\ p_j\ v_j$, it sends f_i to σ_j and marks σ_j as visited by v_j.

(b) Rule 1.6: otherwise, σ_i sends f_i to its parent, p_j, marks σ_j as visited by v_j and generates one g to indicate that σ_i must next clean up useless symbols (rules 2.1–3).

When the source cell, σ_s, as a frontier cell (after receiving a backtrack token), indicated by f, has no more unvisited neighbours, it knows the algorithm termination (rule 1.7).

Partial traces

Table 3.5 shows the partial traces of xP Specification 3.2 for cell σ_4 in Figure 3.4. Omitted symbols $(\dots)$ are $\iota_4\ n_1\ n_2\ n_3\ n_5$ (see Table 3.4).

Table 3.5: Partial traces of xP Specification 3.2 for cell σ_4 in Figure 3.4.

Fig.	Evolution	Content
(a)		$\dots$
(b)	$\{f_5\} \Rightarrow v\ p_5\ v_1\ \{f_4\}_1$	$v\ p_5\ v_1 \dots$
(c)		$v\ p_5\ v_1 \dots$
(d)	$\{f_1\} \Rightarrow\ v_2\ \{f_4\}_2$	$v\ p_5\ v_1\ v_2 \dots$
(e)		$v\ p_5\ v_1\ v_2 \dots$
(f)	$\{f_2\} \Rightarrow\ v_3\ \{f_4\}_3$	$v\ p_5\ v_1\ v_2\ v_3 \dots$
(g)		$v\ p_5\ v_1\ v_2\ v_3 \dots$
(h)	$\{f_3\} \Rightarrow g\ v_5\ \{f_4\}_5$	$v\ p_5\ v_1\ v_2\ v_3\ v_5\ g \dots$

3.3.3 Awerbuch DFS

Awerbuch's algorithm [2] and other more efficient algorithms improve time complexity by having the token traversing tree arcs only: all other arcs are traversed in parallel, by auxiliary messages [3].

In Awerbuch's DFS, each cell knows exactly whether a neighbour is visited, i.e. a neighbour recorded as visited is actually visited.

When a cell is visited for the first time (it was unvisited before receiving the token), it sends its *visited notifications* to all its non-parent neighbours in *parallel* and *waits* until it receives all their acknowledgments. After this, the cell can visit any unvisited neighbour. Thus, a cell does not send a forward token to a visited cell and there is no case (III) as discussed in Section 3.3.

Awerbuch's algorithm works in both asynchronous and synchronous modes; for simplicity, we explain it in a synchronous scenario.

Example 3.4. Figure 3.5 shows one snippet of the evolution of Awerbuch's DFS algorithm in the synchronous mode. Assume that the search started from cell σ_1 and just backtracked to cell σ_5, after building search path $\sigma_1.\sigma_2.\sigma_3.\sigma_5.\sigma_6$.

(a) Cell σ_5 sends its forward token to the only one remaining possibly-unvisited cell, σ_4.

(b) On receiving the forward token from σ_5, unvisted cell σ_4 sets its parent as σ_5, marks itself as visited and sends its visited notification to σ_1, σ_2 and σ_3.

(c) On receiving σ_4's visited notification, cells σ_1, σ_2 and σ_3 mark σ_4 as visited and send back acknowledgments to σ_4.

(d) On receiving all acknowledgments, cell σ_4 sends a backtrack token to its parent, σ_5, because it has no more unvisited neighbours, therefore avoiding probing σ_1, σ_2 and σ_3 needlessly.

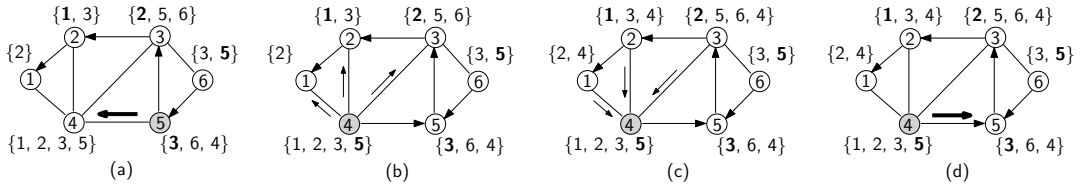


Figure 3.5: An example of Awerbuch's DFS in the synchronous mode. Arcs with arrows: virtual spanning tree child-parent arcs; thick arrows beside arcs: tokens; thin arrows beside arcs: auxiliary messages (visited notifications and acknowledgments); gray cells: frontier cells; elements in braces beside each cell: neighbours recorded as visited, with the parent in bold.

xP Specification 3.3: Awerbuch DFS

Input: Same as in xP Specification 3.2.

Output: Same as in xP Specification 3.2.

Symbols and states

Our xP Specification 3.3 of Awerbuch's DFS uses the same symbols and states as xP Specification 3.2 of the classical DFS, plus a few more specific. Cell σ_i sends u_i as its visited notification and a_i as its acknowledgment; cell σ_i uses w as its counter to wait for acknowledgments and v_j to record a visited neighbour, σ_j.

Consider a cell, σ_i, which receives a token, f_j: (1) a forward token is indicated by $f_j \neg v$, and (2) a backtrack token is indicated by $f_j\ v$ (only unvisited cells receive forward tokens and only visited cells receive backtrack tokens).

The matrix R of xP Specification 3.2 consists of two vectors, informally presented in two groups, according to their functionality and applicability.

1. Main search

1.1. $S_2 \rightarrow_{\texttt{min.min}} S_2\ f_i \mid \iota_i\ s \neg v$

1.2. $S_2\ f_j \rightarrow_{\texttt{min.min}} S_2\ f\ p_j \neg v$

1.3. $S_2\ f_j \rightarrow_{\texttt{min.min}} S_2\ f \mid v$

1.4. $S_2 \rightarrow_{\texttt{max.min}} S_2\ w\ (u_i) \updownarrow_j \mid \iota_i\ f\ n_j \neg v\ p_j$

1.5. $S_2 \rightarrow_{\texttt{min}} S_2\ v \mid f \neg v$

1.6. $S_2\ u_j\ \rightarrow_{\texttt{max.min}}\ S_2\ v_j\ (a_i) \updownarrow_j\ \mid \iota_i$

1.7. $S_2\ a_j\ w\ \rightarrow_{\texttt{max.min}}\ S_2$

1.8. $S_2\ f\ \rightarrow_{\texttt{min.min}}\ S_2\ v_j\ (f_i) \updownarrow_j\ \mid \iota_i\ n_j \neg w\ p_j\ v_j$

1.9. $S_2\ f\ \rightarrow_{\texttt{min.min}}\ S_2\ g\ (f_i) \updownarrow_j\ \mid \iota_i\ p_j \neg w\ s$

1.10. $S_2\ f\ \rightarrow_{\texttt{min}}\ S_2\ g \mid s \neg w$

2. Clean-up

2.1. $S_2\ v_j\ \rightarrow_{\texttt{max.max}}\ S_3 \mid g$

2.2. $S_2\ v\ \rightarrow_{\texttt{max}}\ S_3 \mid g$

2.3. $S_2\ g\ \rightarrow_{\texttt{min}}\ S_3 \mid \neg s$

Initial and final configurations

Table 3.6 shows the initial and final configurations of xP Specification 3.3 for Figure 3.5, highlighting one possible DFS spanning tree. Omitted symbols (...) in the final configuration are the neighbour pointers, n_j's, which are the same as in the initial configuration.

Table 3.6: Initial and final configurations of xP Specification 3.3 for Figure 3.2.

Time	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_2\ s\ \iota_1 n_2 n_4$	$S_2\ \iota_2 n_1 n_3 n_4$	$S_2\ \iota_3 n_2 n_4 n_5 n_6$	$S_2\ \iota_4 n_1 n_2 n_3 n_5$	$S_2\ \iota_5 n_3 n_4 n_6$	$S_2\ \iota_6 n_3 n_5$
23	$S_3\ s\ \iota_1 \boldsymbol{p_1}\ g \ldots$	$S_3\ \iota_2 \boldsymbol{p_1} \ldots$	$S_3\ \iota_3 \boldsymbol{p_2} \ldots$	$S_3\ \iota_4 \boldsymbol{p_5} \ldots$	$S_3\ \iota_5 \boldsymbol{p_3} \ldots$	$S_3\ \iota_6 \boldsymbol{p_5} \ldots$

Rule explanations

The source cell, σ_s, generates one token, f_s, which simulates that σ_s receives a token from a non-existing cell (rule 1.1). Cell σ_i handles the received token, f_j, using the following rules corresponding to the token handling rules discussed in Section 3.3 without case (III).

(I) Rule 1.2: when unvisited σ_i, $\neg v$, receives a forward token, f_j, it sets its parent as p_j and generates one f, indicating it is a frontier cell (σ_i marks itself as visited after sending visited notifications, as later discussed).

(II) Rule 1.3: when visited σ_i, v, receives a backtrack token, f_j, it generates one f.

As a frontier cell, if σ_i is visited for the first time, $f \neg v$, its sends its visited notification, as later discussed. After receiving all acknowledgments, $\neg w$, it can send its token.

A frontier cell that is not visited for the first time immediately sends its token.

(a) Rule 1.8: if σ_i has any unvisited non-parent neighbour, $n_j \neg p_j\ v_j$, it sends f_i to σ_j and marks σ_j as visited by v_j.

(b) Rule 1.9: otherwise, σ_i sends f_i to its parent, p_j, and generates one g to indicate that σ_i must next clean up useless symbols (rules 2.1–3).

The following rules are used for sending visited notifications and receiving acknowledgments.

- If cell σ_i is visited for the first time, $f \neg v$, it sends its visited notification, u_i, to all its non-parent neighbours, $n_j \neg p_j$, and generates one w for each receiver (rule 1.4); then σ_i marks itself as visited by v (rule 1.5).
- On receiving u_i from σ_i, cell σ_j transforms u_i into v_i and sends an acknowledgment, a_j, to σ_i (rule 1.6).
- For each received a_j, cell σ_i deletes one w (rule 1.7); when σ_i receives all acknowledgments, $\neg w$, it can send its token, as discussed in (a) and (b) above.

When the source cell, σ_s, as a frontier cell (after receiving a backtrack token), indicated by f, has no more unvisited neighbours, it knows the algorithm termination (rule 1.10).

Partial traces

Table 3.7 shows the partial traces of xP Specification 3.3 for cell σ_4 in Figure 3.5. Omitted symbols $(\dots)$ are $\iota_4\ n_1\ n_2\ n_3\ n_5$ (see Table 3.6).

Table 3.7: Partial traces of xP Specification 3.3 for cell σ_4 in Figure 3.5.

Fig.	Evolution	Content
(a)		$v_1\ v_2\ v_3\ v_5 \dots$
(b)	$\{f_5\} \Rightarrow v\ f\ p_5\ w^3\ \{u_4\}_{1,2,3}$	$v_1\ v_2\ v_3\ v_5\ v\ f\ p_5\ w^3 \dots$
(c)		$v_1\ v_2\ v_3\ v_5\ v\ f\ p_5\ w^3 \dots$
(d)	$\{a_1\ a_2\ a_3\}\ w^3\ f \Rightarrow g\ \{f_4\}_5$	$v_1\ v_2\ v_3\ v_5\ v\ p_5\ g \dots$

3.3.4 Cidon DFS

Cidon's algorithm [13] improves Awerbuch's algorithm (§ 3.3.3) by not using acknowledgments, therefore removing a delay [3]. A cell that is visited for the first time sends its visited notifications to all its non-parent neighbours at the *same time* while sending its token to a neighbour.

Each cell records a list of *possibly-unvisited* neighbours, i.e. neighbour cells that are not known as having been visited.

In this algorithm, a cell may send a forward token to a visited cell if it has not yet received the cell's visited notification. To solve this problem, each visited cell records the last cell to which it has sent a *forward* token, as its *most-recently-used (MRU) cell*, σ_k.

- When σ_i receives a token from σ_j, if $\sigma_j \neq \sigma_k$, this is a forward token and σ_i interprets the token from σ_j as σ_j's visited notification (if $\sigma_j = \sigma_k$, this is a backtrack token from σ_j).
- When σ_i receives σ_j's visited notification, if $\sigma_j = \sigma_k$, σ_i interprets the visited notification from σ_j as a backtrack token (if $\sigma_j \neq \sigma_k$, this is just a visited notification from σ_j).

Cidon's algorithm works in both asynchronous and synchronous modes; we explain it in both asynchronous and synchronous scenarios. Later, we use a synchronous version of Cidon's DFS in Sections 4.3.4–4.3.6 and 4.4.1.

Example 3.5. Figure 3.6 illustrates one snippet of the evolution of Cidon's DFS algorithm in the synchronous mode. The search started from cell σ_1 and has just backtracked to cell σ_5, after building search path $\sigma_1.\sigma_2.\sigma_3.\sigma_5.\sigma_6$.

(a) Cell σ_5 sends its forward token to the only one possibly-unvisited cell, σ_4.

(b) On receiving the forward token from σ_5, unvisited cell σ_4 sets its parent as σ_5, marks itself as visited and sends its visited notifications to σ_1, σ_2 and σ_3. At the same time, σ_4 sends its backtrack token to its parent, σ_5, because it has no more possibly-unvisited neighbours.

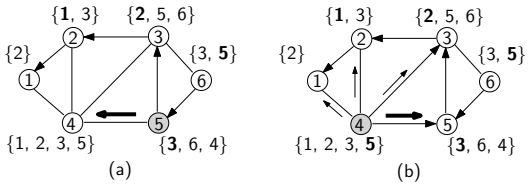


Figure 3.6: An example of Cidon's DFS algorithm in the synchronous mode. Arcs with arrows: virtual spanning tree child-parent arcs; thick arrows beside arcs: tokens; thin arrows beside arcs: auxiliary messages (visited notifications); gray cells: frontier cells; elements in braces beside each cell: neighbours recorded as visited, with the parent in bold.

Example 3.6. Figure 3.7 shows one snippet of the evolution of Cidon's DFS algorithm in one possible asynchronous run. Assume that the search started from cell σ_1 and just backtracked to cell σ_5, after building search path $\sigma_1.\sigma_2.\sigma_3.\sigma_5.\sigma_6$. Moreover, σ_2's visited notification has not yet arrived at σ_4 and thus σ_4 only knows that σ_1, σ_3 and σ_5 are visited.

(a) Cell σ_5 sends its forward token to the only one possibly-unvisited cell, σ_4.

(b) On receiving the forward token from σ_5, unvisited cell σ_4 sets its parent as σ_5, marks itself as visited and sends its visited notifications to σ_1, σ_2 and σ_3. At the same time, σ_4 sends its forward token to σ_2 and records its MRU cell as σ_2 because it has not yet received σ_2's visited notification.

(c) On receiving the token from σ_4, visited cell σ_2 finds that σ_4 is not its MRU cell, σ_3, so σ_2 simply records σ_4 as visited.

Assume that σ_2's visited notification arrives at σ_4.

(d) On receiving σ_2's visited notification, σ_4 finds that σ_2 is its MRU cell, so it interprets this as a backtrack token from σ_2 and sends a backtrack token to its parent, σ_5, because it has no more unvisited neighbours.

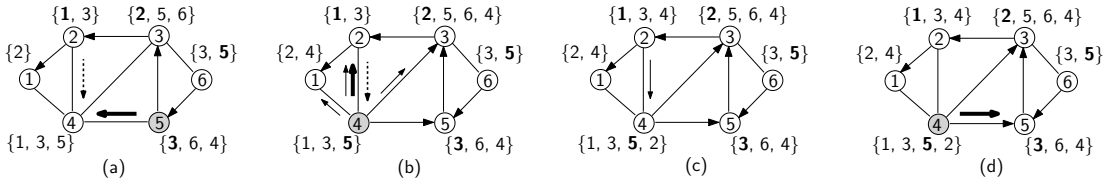


Figure 3.7: An example of Cidon's DFS in one possible asynchronous run. Arcs with arrows: virtual spanning tree child-parent arcs; thick arrows beside arcs: tokens; thin arrows beside arcs: auxiliary messages (visited notifications); gray cells: frontier cells; elements in braces beside each cell: neighbours recorded as visited, with the parent in bold; dotted arrows: σ_2's visited notifications that have not yet arrived at σ_4.

xP Specification 3.4: Cidon DFS

Input: Same as in xP Specification 3.2.

Output: Same as in xP Specification 3.2.

Symbols and states

Our xP Specification 3.4 of Cidon's DFS uses the same symbols and states as xP Specification 3.3 of Awerbuch's DFS, plus one symbol, m_j, which is used to record a cell's MRU cell.

Consider a cell, σ_i, which records its MRU cell as σ_k and receives a token, f_j, from σ_j: (1) a forward token is indicated by (a) $f_j \neg v$ or (b) $f_j\ v \neg m_j$ (as discussed before, $\sigma_j \neq \sigma_k$), and (2) a backtrack token is indicated by $f_j\ m_j\ v$ (as discussed before, $\sigma_j = \sigma_k$).

The matrix R of xP Specification 3.2 consists of two vectors, informally presented in two groups, according to their functionality and applicability.

1. Main search

1.1. $S_2 \ \rightarrow_{\texttt{min.min}} \ S_2 \ f_i \mid \iota_i \ s \neg v$

1.2. $S_2 \ f_j \ \rightarrow_{\texttt{min.min}} \ S_2 \ f \ p_j \neg v$

1.3. $S_2 \ u_j \ \rightarrow_{\texttt{min.min}} \ S_2 \ f \mid m_j$

1.4. $S_2 \ f_j \ \rightarrow_{\texttt{min.min}} \ S_2 \ f \mid m_j \ v$

1.5. $S_2 \ f_j \ \rightarrow_{\texttt{min.min}} \ S_2 \ v_j \mid v \neg m_j$

1.6. $S_2 \ \rightarrow_{\texttt{max.min}} \ S_2 \ (u_i) \updownarrow_j \ \mid \iota_i \ f \ n_j \neg v \ p_j$

1.7. $S_2 \ \rightarrow_{\texttt{min}} \ S_2 \ v \mid f \neg v$

1.8. $S_2 \ u_j \ \rightarrow_{\texttt{max.min}} \ S_2 \ v_j$

1.9. $S_2 \ m_j \ \rightarrow_{\texttt{min.min}} \ S_2 \mid f$

1.10. $S_2 \ f \ \rightarrow_{\texttt{min.min}} \ S_2 \ m_j \ v_j \ (f_i) \updownarrow_j \ \mid \iota_i \ n_j \neg p_j \ v_j$

1.11. $S_2 \ f \ \rightarrow_{\texttt{min.min}} \ S_2 \ g \ (f_i) \updownarrow_j \ \mid \iota_i \ p_j \neg s$

1.12. $S_2 \ f \ \rightarrow_{\texttt{min}} \ S_2 \ g \mid s$

2. Clean-up

2.1. $S_2 \ v_j \ \rightarrow_{\texttt{max.max}} \ S_3 \mid g$

2.2. $S_2 \ v \ \rightarrow_{\texttt{max}} \ S_3 \mid g$

2.3. $S_2 \ g \ \rightarrow_{\texttt{min}} \ S_3 \mid \neg s$

Initial and final configurations

Table 3.8 shows the initial and final configurations of xP Specification 3.4 for Figures 3.6 ("Synch." row) and 3.7 ("Asynch." row), highlighting one possible DFS spanning tree. Omitted symbols (...) in the final configuration are the neighbour pointers, n_j's, which are the same as in the initial configuration.

Table 3.8: Initial and final configurations of xP Specification 3.4 for Figures 3.6 and 3.7.

Time	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_2 \ s \ \iota_1 n_2 n_4$	$S_2 \ \iota_2 n_1 n_3 n_4$	$S_2 \ \iota_3 n_2 n_4 n_5 n_6$	$S_2 \ \iota_4 n_1 n_2 n_3 n_5$	$S_2 \ \iota_5 n_3 n_4 n_6$	$S_2 \ \iota_6 n_3 n_5$
Synch. 11	$S_3 \ s \ \iota_1 \boldsymbol{p_1} \ g \ldots$	$S_3 \ \iota_2 \boldsymbol{p_1} \ldots$	$S_3 \ \iota_3 \boldsymbol{p_2} \ldots$	$S_3 \ \iota_4 \boldsymbol{p_5} \ldots$	$S_3 \ \iota_5 \boldsymbol{p_3} \ldots$	$S_3 \ \iota_6 \boldsymbol{p_5} \ldots$
Asynch. 6	$S_3 \ s \ \iota_1 \boldsymbol{p_1} \ g \ldots$	$S_3 \ \iota_2 \boldsymbol{p_1} \ldots$	$S_3 \ \iota_3 \boldsymbol{p_2} \ldots$	$S_3 \ \iota_4 \boldsymbol{p_5} \ldots$	$S_3 \ \iota_5 \boldsymbol{p_3} \ldots$	$S_3 \ \iota_6 \boldsymbol{p_5} \ldots$

Rule explanations

The source cell, σ_s, generates one token, f_s, which simulates that σ_s receives a token from a non-existing cell (rule 1.1). Cell σ_i handles the received token, f_j, using the following rules corresponding to the token handling rules discussed in Section 3.3.

The additional case (II') interprets the visited notification from σ_j as a backtrack token and case (III) interprets the token from σ_j as σ_j's visited notification, as discussed before.

(I) Rules 1.2: when unvisited σ_i, $\neg\, v$, receives a forward token, f_j, it sets its parent as p_j and generates one f, indicating it is a frontier cell.

(II) Rule 1.4: when visited σ_i, v, receives a backtrack token, $f_j\ m_j$, it generates one f.

(II') Rule 1.3: when visited σ_i, v, receives visited notification u_j from its MRU cell, m_j, it interprets u_j as a backtrack token and thus generates one f.

(III) Rule 1.5: when visited σ_i, v, receives a forward token, f_j, from a cell that is not its MRU cell, $\neg\, m_j$, it interprets f_j as a visited notification by v_j.

As a frontier cell, if σ_i is visited for the first time, $f\ \neg\ v$, it sends its visited notification, u_i, to all its non-parent neighbours, $n_j\ \neg\ p_j$ (rule 1.6); each visited notification receiver transforms u_i into v_i (rule 1.8). Then, σ_i marks itself as visited (rule 1.7) and sends its token in the same step.

A frontier cell that is not visited for the first time directly sends its token.

(a) Rule 1.10: if σ_i has any unvisited non-parent neighbour, $n_j\ \neg\ p_j\ v_j$, it sends f_i to σ_j, marks σ_j as visited by v_j and records its MRU cell as m_j.

(b) Rule 1.11: otherwise, σ_i sends f_i to its parent, p_j, and generates one g to indicate that σ_i must next clean up useless symbols (rules 2.1–3).

When the source cell, σ_s, as a frontier cell (after receiving a backtrack token), indicated by f, has no more unvisited neighbours, it knows the algorithm termination (rule 1.12).

Partial traces

Table 3.9 show the partial traces of xP Specification 3.4 for cell σ_4 in Figure 3.7 in one possible asynchronous run. Omitted symbols $(\ldots)$ are $\iota_4\ n_1\ n_2\ n_3\ n_5$ (see Table 3.8).

3.3.5 Sharma DFS

Sharma et al.'s algorithm [61] further improves time complexity, at the cost of increasing the message size, by sending a *list* of visited cells together with the *token* [3], therefore eliminating unnecessary message exchanges to inform neighbours of visited

Table 3.9: Partial traces of xP Specification 3.4 for cell σ_4 in Figure 3.6 in one possible asynchronous run.

Fig.	Evolution	Content
(a)		$v_1\ v_3\ v_5 \ldots$
(b)	$\{f_5\} \Rightarrow v\ p_5\ m_2\ \{u_4\}_{1,2,3}\ \{f_4\}_2$	$v_1\ v_2\ v_3\ v_5\ v\ p_5\ m_2 \ldots$
(c)		$v_1\ v_2\ v_3\ v_5\ v\ p_5\ m_2 \ldots$
(d)	$\{u_2\}\ m_2 \Rightarrow g\ \{f_4\}_5$	$v_1\ v_2\ v_3\ v_5\ v\ p_5\ g \ldots$

status. The visited list is augmented in each cell that is visited for the first time and grows up to the size of n, where n is the total number of cells.

In this algorithm, there is no possible-unvisited cell that is actually visited: a cell knows exactly whether a neighbour has been visited or not. Thus, there is no case (III) as discussed in Section 3.3.1.

Sharma et al.'s algorithm works in both asynchronous and synchronous modes; for simplicity, we explain it in a synchronous scenario.

Example 3.7. Figure 3.8 shows one snippet of the evolution of Sharma's DFS algorithm in the synchronous mode. Assume that the search started from cell σ_1 and just arrived at cell σ_5, after building search path $\sigma_1.\sigma_2.\sigma_3.\sigma_5$.

(a) Cell σ_5 sends its visited list, {1, 2, 3, 5}, with its forward token, to one of the unvisited neighbours, σ_6.

(b) On receiving the forward token and visited list from σ_5, unvisited cell σ_6 sets its parent as σ_5, marks itself as visited, adds its own mark to the visited list and sends the list, {1, 2, 3, 5, 6}, with its backtrack token to its parent, σ_5, because it has no more unvisited neighbours.

(c) On receiving the backtrack token and visited list from σ_6, visited cell σ_5 forwards the list, {1, 2, 3, 5, 6}, with its forward token to the only one unvisited neighbour, σ_4.

(d) On receiving the forward token and visited list from σ_5, unvisited cell σ_4 sets its parent as σ_5, marks itself as visited, adds its own mark to the visited list and sends the list, {1, 2, 3, 5, 6, 4}, with its backtrack token to its parent, σ_5, because it has no more unvisited neighbours.

xP Specification 3.5: Sharma DFS

Input: Same as in xP Specification 3.2.

Output: Same as in xP Specification 3.2.

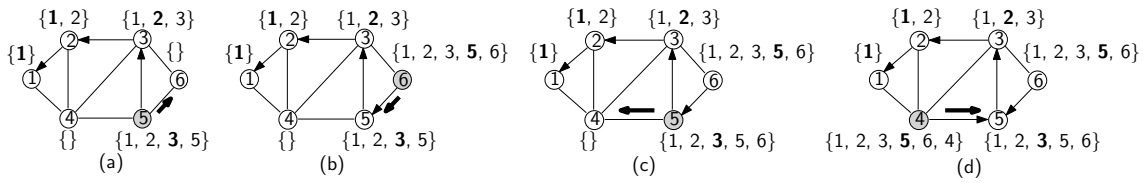


Figure 3.8: An example of Sharma et al.'s DFS in the synchronous mode. Arcs with arrows: virtual spanning tree child-parent arcs; thick arrows beside arcs: messages; gray cells: frontier cells; elements in braces beside each cell: neighbours recorded as visited, with the parent in bold.

Symbols and states

Our xP Specification 3.5 of Sharma et al.'s DFS uses the same symbols and states as xP Specification 3.2 of the classical DFS, plus a few more specific. Cell σ_i uses v_j as a *visited mark* of cell σ_j in the visited list and t_j to indicate the *token target* cell to which it must next send its token and visited list.

Consider a cell, σ_i, which receives a token, f_j: (1) a forward token is indicated by $f_j \neg v$, and (2) a backtrack token is indicated by $f_j\ v$ (only unvisited cells receive forward tokens and only visited cells receive backtrack tokens).

The matrix R of xP Specification 3.2 consists of two vectors, informally presented in two groups, according to their functionality and applicability.

1. Main search

1.1. $S_2 \rightarrow_{\texttt{min.min}} S_2\ f_i \mid \iota_i\ s \neg v$

1.2. $S_2\ f_j \rightarrow_{\texttt{min.min}} S_2\ f\ v\ v_i\ p_j \mid \iota_i \neg v$

1.3. $S_2\ f_j \rightarrow_{\texttt{min.min}} S_2\ f \mid v$

1.4. $S_2\ v_k \rightarrow_{\texttt{max.max}} S_2 \mid f\ v_k$

1.5. $S_2\ f \rightarrow_{\texttt{min.min}} S_2\ t_j \mid n_j \neg p_j\ v_j$

1.6. $S_2\ f \rightarrow_{\texttt{min.min}} S_2\ g\ t_j \mid p_j \neg s$

1.7. $S_2\ f \rightarrow_{\texttt{min}} S_2\ g \mid s$

1.8. $S_2\ v_k \rightarrow_{\texttt{max.min}} S_2\ v_k\ (v_k) \updownarrow_j \mid t_j$

1.9. $S_2\ t_j \rightarrow_{\texttt{min.min}} S_2\ (f_i) \updownarrow_j \mid \iota_i$

2. Clean-up

2.1. $S_2\ v_j \rightarrow_{\texttt{max.max}} S_3 \mid g$

2.2. $S_2\ v \rightarrow_{\texttt{max}} S_3 \mid g$

2.3. $S_2\ g \rightarrow_{\texttt{min}} S_3 \mid \neg s$

Initial and final configurations

Table 3.10 shows the initial and final configurations of xP Specification 3.5 for Figure 3.8, highlighting one possible DFS spanning tree. Omitted symbols (...) in the final configuration are the neighbour pointers, n_j's, which are the same as in the initial configuration.

Table 3.10: Initial and final configurations of xP Specification 3.5 for Figure 3.2.

Time	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_2\ s\ \iota_1 n_2 n_4$	$S_2\ \iota_2 n_1 n_3 n_4$	$S_2\ \iota_3 n_2 n_4 n_5 n_6$	$S_2\ \iota_4 n_1 n_2 n_3 n_5$	$S_2\ \iota_5 n_3 n_4 n_6$	$S_2\ \iota_6 n_3 n_5$
11	$S_3\ s\ \iota_1 \boldsymbol{p_1}\ g \ldots$	$S_3\ \iota_2 \boldsymbol{p_1} \ldots$	$S_3\ \iota_3 \boldsymbol{p_2} \ldots$	$S_3\ \iota_4 \boldsymbol{p_5} \ldots$	$S_3\ \iota_5 \boldsymbol{p_3} \ldots$	$S_3\ \iota_6 \boldsymbol{p_5} \ldots$

Rule explanations

The source cell, σ_s, generates one token, f_s, which simulates that σ_s receives a token from a non-existing cell (rule 1.1). Cell σ_i handles the received token, f_j, using the following rules corresponding to the token handling rules discussed in Section 3.3.1 without case (III).

(I) Rule 1.2: when unvisited σ_i, $\neg\ v$, receives a forward token, f_j, it sets its parent as p_j, marks itself as visited by v, adds its own mark, v_i, to its visited list and generates one f, indicating it is a frontier cell.

(II) Rule 1.3: when visited σ_i, v, receives a backtrack token, f_j, it generates one f.

For the visited list, v_k's, received with the token in (I) and (II), cell σ_i only keeps one copy of each visited mark (it may contain the same visited mark of the old visited list) (rule 1.4).

As a frontier, σ_i sends its token as follows.

(a) Rules 1.5: if σ_i has any unvisited non-parent neighbour, $n_j\ \neg\ p_j\ v_j$, it decides its token target, t_j.

(b) Rules 1.6: otherwise, σ_i decides its token target, t_j, as its parent, p_j, and generates one g to indicate that σ_i must next clean up useless symbols (rules 2.1–3).

Then cell σ_i sends the visited list, v_k's (rule 1.8) with the token (rule 1.9) to its token target, t_j.

When the source cell, σ_s, as a frontier cell (after receiving a backtrack token), indicated by f, has no more unvisited neighbours, it knows the algorithm termination (rule 1.7).

Partial traces

Table 3.11 shows the partial traces of xP Specification 3.5 for cell σ_5 in Figure 3.8. Omitted symbols (...) are $\iota_5\ n_3\ n_4\ n_6$ (See Table 3.10).

Table 3.11: Partial traces of xP Specification 3.5 for cell σ_5 in Figure 3.8.

Fig.	Evolution	Content
(a)	$\{f_3\ v_1\ v_2\ v_3\} \Rightarrow v\ p_3\ v_1\ v_2\ v_3\ v_5\ \{f_5\ v_1\ v_2\ v_3\ v_5\}_6$	$v\ p_3\ v_1\ v_2\ v_3\ v_5 \ldots$
(b)		$v\ p_3\ v_1\ v_2\ v_3\ v_5 \ldots$
(c)	$\{f_6\ v_1\ v_2\ v_3\ v_5\ v_6\}\ v_1\ v_2\ v_3\ v_5 \Rightarrow$ $v_1\ v_2\ v_3\ v_5\ v_6\ \{f_5\ v_1\ v_2\ v_3\ v_5\ v_6\}_4$	$v\ p_3\ v_1\ v_2\ v_3\ v_5\ v_6 \ldots$
(d)		$v\ p_3\ v_1\ v_2\ v_3\ v_5\ v_6 \ldots$

3.3.6 Makki DFS

Makki et al.'s algorithm [43] improves Sharma et al.'s algorithm by using a *dynamic backtracking* technique. There is also no case (III) as discussed in Section 3.3.1.

Each non-source cell keeps track of the most recent global *split point* and records its *return cell*, i.e. the lowest ancestor cell, which may still have unvisited neighbours. When the search backtracks from cell σ_i, if the cell has a non-tree edge to its return cell, σ_j, it backtracks directly to σ_j via that edge, rather than following the longer DFS tree branch to σ_j [3].

Initially, the split point is the source cell, $splitpoint = \sigma_s$. A cell σ_i that is visited for the first time records its return cell as its received split point, $\sigma_i.returncell = splitpoint$. The split point is dynamically updated in each frontier cell that has one or more unvisited neighbours and is sent with the forward token. Consider a frontier cell, σ_i.

- If σ_i has one or more unvisited neighbours, it updates the split point and sends the split point with its forward token:
  1. $splitpoint = \sigma_i$, if σ_i has two or more unvisited neighbours;
  2. $splitpoint = \sigma_i.returncell$, if σ_i has only one unvisited neighbour.
- Otherwise, if σ_i has no more unvisited neighbours, it backtracks without sending the split point.

Makki et al.'s algorithm works in both asynchronous and synchronous modes; for simplicity, we explain it in a synchronous scenario.

Example 3.8. Figure 3.9 shows one snippet of the evolution of Makki et al.'s algorithm in the synchronous mode. Assume that the search started from cell σ_1 and just arrived at cell σ_5, after building search path $\sigma_1.\sigma_2.\sigma_3.\sigma_5$.

(a) Cell σ_5 has two unvisited neighbours, σ_4 and σ_6, so it sends $splitpoint = \sigma_5$ and the visited list, $\{1, 2, 3, 5\}$, with its forward token, to one of the unvisited neighbours, σ_6.

(b) On receiving the forward token, visited list and split point from σ_5, unvisited cell σ_6 sets its parent as σ_5, marks itself as visited, adds its own mark to the visited list and records $\sigma_6.returncell = \sigma_5$. Cell σ_6 has no unvisited neighbour, so it sends the visited list, $\{1, 2, 3, 5, 6\}$, with its backtrack token, to its return cell, σ_5.

(c) On receiving the backtrack token and visited list from σ_6, because cell σ_5 has only one unvisited neighbour, σ_4, it sends $splitpoint = \sigma_5.returncell = \sigma_3$ and the visited list, $\{1, 2, 3, 5, 6\}$, with its forward token to σ_4.

(d) On receiving the forward token, visited list and split point from σ_5, unvisited cell σ_4 sets its parent as σ_5, marks itself as visited, adds its own mark to the visited list and records $\sigma_4.returncell = splitpoint = \sigma_3$. Cell σ_4 has no more unvisited neighbours, so it sends the list, $\{1, 2, 3, 5, 6, 4\}$, with its backtrack token to its return cell, σ_3, via a direct edge, rather than following the longer tree path, $\sigma_4.\sigma_5.\sigma_3$.

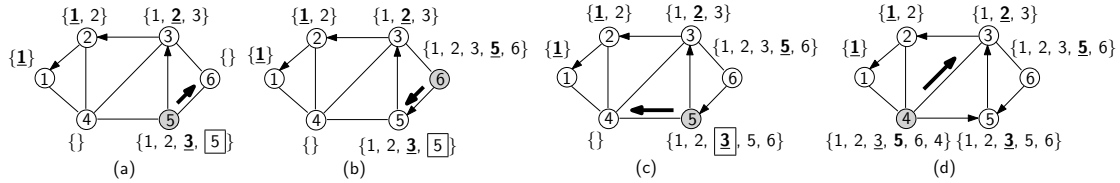


Figure 3.9: An example of Makki et al.'s DFS in the synchronous mode. Arcs with arrows: virtual spanning tree child-parent arcs; thick arrows near arcs: messages; elements in braces beside each cell: neighbours recorded as visited, with its parent in bold, its return cell underlined and the split point boxed.

xP Specification 3.6: Makki DFS

Input: Same as in xP Specification 3.2.

Output: Same as in xP Specification 3.2, except that the non-backtracked cells (due to the dynamic backtracking technique) end in state S_2 (if needed, all cells can end in the same state, S_3, by using a supplemented broadcast phase).

Symbols and states

Our xP Specification 3.6 of Makki et al.'s DFS uses the same symbols and states as xP Specification 3.5 of Sharma et al.'s DFS, plus a few more specific: cell σ_i uses r_j to record its return cell and sends y_k as the split point.

The matrix R of xP Specification 3.2 consists of two vectors, informally presented in two groups, according to their functionality and applicability.

1. Main search

1.1. $S_2 \rightarrow_{\texttt{min.min}} S_2\ f_i\ y_i \mid \iota_i\ s \neg v$

1.2. $S_2\ f_j\ y_k \rightarrow_{\texttt{min.min}} S_2\ f\ v\ v_i\ p_j\ r_k \mid \iota_i \neg v$

1.3. $S_2\ f_j \rightarrow_{\texttt{min.min}} S_2\ f \mid v$

1.4. $S_2\ v_k \rightarrow_{\texttt{max.max}} S_2 \mid f\ v_k$

1.5. $S_2\ f\ n_j \rightarrow_{\texttt{min.min}} S_2\ t_j \mid \neg\ p_j\ v_j$

1.6. $S_2 \rightarrow_{\texttt{min.min}} S_2\ y_i \mid \iota_i\ n_k\ t_j \neg\ p_k\ v_k$

1.7. $S_2 \rightarrow_{\texttt{min.min}} S_2\ y_k \mid \iota_i\ r_k\ t_j \neg\ y_i$

1.8. $S_2 \rightarrow_{\texttt{min.min}} S_2\ n_j \mid t_j$

1.9. $S_2\ f \rightarrow_{\texttt{min.min}} S_2\ g\ t_j \mid r_j\ n_j \neg\ s$

1.10. $S_2\ f \rightarrow_{\texttt{min.min}} S_2\ g\ t_j \mid p_j\ r_k \neg\ s\ n_k$

1.11. $S_2\ f \rightarrow_{\texttt{min}} S_2\ g \mid s$

1.12. $S_2\ v_k \rightarrow_{\texttt{max.min}} S_2\ v_k\ (v_k) \updownarrow_j \mid t_j$

1.13. $S_2\ t_j \rightarrow_{\texttt{min.min}} S_2\ (f_i) \updownarrow_j \mid \iota_i$

1.14. $S_2\ y_k \rightarrow_{\texttt{min.min}} S_2\ (y_k) \updownarrow_j \mid t_j$

2. Clean-up

2.1. $S_2\ v_j \rightarrow_{\texttt{max.max}} S_3 \mid g$

2.2. $S_2\ r_j \rightarrow_{\texttt{max.max}} S_3 \mid g$

2.3. $S_2\ v \rightarrow_{\texttt{max}} S_3 \mid g$

2.4. $S_2\ g \rightarrow_{\texttt{min}} S_3 \mid \neg\ s$

Initial and final configurations

Table 3.12 shows the initial and final configurations of xP Specification 3.6 for Figure 3.9, highlighting one possible DFS spanning tree. In the final configuration, omitted symbols (...) are the neighbour pointers, n_j's, which are the same as in the initial configuration.

Table 3.12: Initial and final configurations of xP Specification 3.6 for Figure 3.2.

Time	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
0	$S_2\ s\ \iota_1 n_2 n_4$	$S_2\ \iota_2 n_1 n_3 n_4$	$S_2\ \iota_3 n_2 n_4 n_5 n_6$	$S_2\ \iota_4 n_1 n_2 n_3 n_5$	$S_2\ \iota_5 n_3 n_4 n_6$	$S_2\ \iota_6 n_3 n_5$
10	$S_3\ s\ \iota_1 \boldsymbol{p_1}\ g \ldots$	$S_3\ \iota_2 \boldsymbol{p_1} \ldots$	$S_3\ \iota_3 \boldsymbol{p_2} \ldots$	$S_3\ \iota_4 \boldsymbol{p_5} \ldots$	$S_2\ \iota_5 \boldsymbol{p_3} \ldots$	$S_3\ \iota_6 \boldsymbol{p_5} \ldots$

Rule explanations

Because Makki et al.'s DFS builds on Sharma et al.'s DFS (§ 3.3.5), we only explain the rules of the dynamic backtracking technique.

The source cell, σ_s, generates one token, f_s, and one split point symbol, y_s, which simulates that σ_s receives a token and a split point from a non-existing cell (rule 1.1). Cell σ_i handles the received token, f_j, using the following rules corresponding to the token handling rules discussed in Section 3.3.1 without case (III).

(I) Rule 1.2: when unvisited cell σ_i, $\neg\, v$, receives the split point, y_k, with a forward token, f_j, it records its return cell as r_k.

(II) Rule 1.3: when visited cell σ_i, v, receives a backtrack token, f_j, it does not receive the split point (the split point is only sent with the forward token).

As a frontier, σ_i decides its token target cell; if possible, σ_a also generates the split point symbol.

(a) Rule 1.5: if σ_i has one or more unvisited neighbours, $n_j\ \neg\ v_j$, it decides its token target cell, t_j, by transforming n_j into t_j (rule 1.5), and generates a split point symbol:

- if σ_i has two or more unvisited neighbour, $n_k\ \neg\ p_k\ v_k$, it generates a split point symbol, y_i (rule 1.6);
- otherwise, if σ_i has only one unvisited neighbour, $\neg\ y_i$, it generates y_k using r_k (rule 1.7).

After determining the split point, σ_i restores n_j using t_j (rule 1.8).

(b) Rules 1.9–10: otherwise, if σ_i has no unvisited neighbour, it decides its target cell, t_j:

- if σ_i's return cell is its neighbour, $r_j\ n_j$, it generates one t_j (rule 1.9);
- otherwise, $r_k\ \neg\ n_k$, σ_i generates t_j using p_j (rule 1.10).

Then cell σ_i sends the split point, y_k, if any, with its token and visited list to its token target, t_j (rules 1.12–14).

When the source cell, σ_s, as a frontier cell (after receiving a backtrack token), indicated by f, has no more unvisited neighbours, it knows the algorithm termination (rule 1.11).

Partial traces

Table 3.13 shows the partial traces of xP Specification 3.6 for cell σ_4 in Figure 3.9. Omitted symbols $(\dots)$ are $\iota_4\ n_1\ n_2\ n_3\ n_5$ (see Table 3.12).

Table 3.13: Partial traces of xP Specification 3.6 for cell σ_4 in Figure 3.9.

Fig.	Evolution	Content
(a) (b) (c)		...
(d)	$\{f_5\ y_3\ v_1\ v_2\ v_3\ v_5\ v_6\} \Rightarrow$ $v\ p_5\ v_1\ v_2\ v_3\ v_5\ v_6\ v_4\ r_3\ g\ \{f_4\ v_1\ v_2\ v_3\ v_5\ v_6\ v_4\}_3$	$v\ p_5\ v_1\ v_2\ v_3\ v_5\ v_6\ v_4$ $r_3\ g \ldots$

3.4 Distributed Breadth-First Search Algorithms

BFS is a fundamental technique, apparently inherently *parallel*. There are a number of distributed BFS algorithms, which speed up BFS in distributed systems, such as Synchronous BFS (SynchBFS), Asynchronous BFS (AsynchBFS), an improved Asynchronous BFS with known graph diameter, Layered BFS and Hybrid BFS [42]. This section discusses SynchBFS and AsynchBFS. Figure 3.11 (c) shows a virtual BFS spanning tree, T, in an *undirected* graph, G.

3.4.1 BFS Token Handling Rules in Undirected Graphs

BFS explores as many branches as possible concurrently, which explores by *visit tokens*.

BFS starts by the source cell's broadcasting its visit token to all neighbours. A cell handles its received visit tokens based on the following *token handling rules*.

(I) When an *unvisited* cell, σ_i, receives visit tokens from its neighbour cells, it selects *one* of the sending cells, σ_j, as its *parent* and marks itself as visited, becoming a *frontier* cell. Then:

  (a) it sends its visit token to all its non-parent neighbours, if any.

(II) When *visited* cell, σ_i, receives visit tokens from its neighbour cells, it ignores the visit tokens.

Example 3.9. Figure 3.10 shows how a cell handles its received visit tokens based on the token handling rules, where cell σ_2 becomes a frontier cell in case (Ia).

(Ia) When an *unvisited* cell, σ_2, receives visit tokens from its neighbour cells, σ_1 and σ_5, it selects *one* of the sending cells, σ_1, as its *parent* and marks itself as visited, becoming a *frontier* cell. Then σ_2 sends its visit token to all its non-parent neighbours, σ_3, σ_4 and σ_5.

(II) When *visited* cell, σ_2, receives visit tokens from its neighbour cells, σ_4 and σ_5, it ignores the visit tokens.

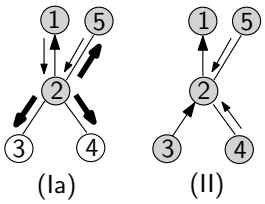


Figure 3.10: A cell, σ_2, handles its received visit tokens in distributed BFS. Edges with arrows: virtual spanning tree child-parent arcs; thin arrows beside edges: visit tokens received by σ_2; thick arrows beside edges: visit tokens sent by σ_2; gray cells: visited cells.

As mentioned in Section 3.2, distributed BFS algorithms also have the race condition problem, which can be solved by using the `min.min` mode (see rule 1.2 of xP Specification 3.7 for an example), in the same way as the Echo algorithm.

For all BFS algorithms in this section, no cell knows when the algorithm terminates. If this is required, we can apply a termination detection algorithm to this algorithm, as later discussed in Section 3.6.

Following our joint work [3], this section first describes how SynchBFS succeeds in the synchronous mode but *fails* in the asynchronous mode and then provides its xP solution. Next, we discusses the AsynchBFS algorithm, which works correctly in both synchronous and asynchronous modes, and presents its xP specification.

3.4.2 Synchronous BFS (SynchBFS)

Synchronous BFS (SynchBFS) algorithm [42] is a synchronous algorithm, which produces a BFS spanning tree in the synchronous mode.

Initially, the source cell broadcasts a visit token. On receiving the visit token, an unvisited cell marks itself as visited and chooses *one* of the token sending cells as its parent. Next, it sends its visit token to all non-parent neighbours [3]. In the synchronous mode, the algorithm terminates when no more visit tokens are sent; however, no cell knows when the algorithm terminates.

We first explain SynchBFS in a synchronous scenario and then show how it fails in an asynchronous scenario.

Example 3.10. Figure 3.11 illustrates how SynchBFS works in the *synchronous* mode.

(a) At the start, the source cell, σ_1, broadcasts its visit token.

(b) On receiving the visit token from σ_1, each of the unvisited cells, σ_2, σ_3 and σ_4, marks itself as visited, sets its parent as σ_1 and sends its visit tokens to all non-parent neighbours.

(c) On receiving visit tokens, visited cells σ_2, σ_3 and σ_4 ignore the visit tokens.

Finally, the algorithm terminates because no more tokens are sent, determining a BFS spanning tree.

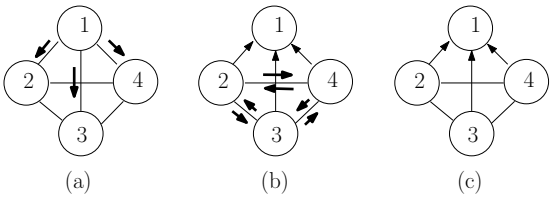


Figure 3.11: An example of SynchBFS in the synchronous mode. Arcs with arrows: virtual spanning tree child-parent arcs; thick arrows near arcs: visit tokens. The result (c) is a BFS spanning tree.

However, in the asynchronous mode, SynchBFS does not necessarily build a BFS spanning tree. Figure 3.12 shows one possible evolution of SynchBFS in one possible asynchronous run, where the resulting spanning tree (f) is *not* a BFS spanning tree.

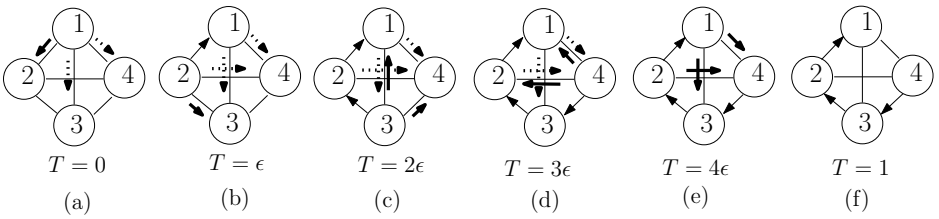


Figure 3.12: An example of SynchBFS in one possible asynchronous run. Arcs with arrows: virtual spanning tree child-parent arcs; thick arrows near arcs: visit tokens; dotted arrows: visit tokens still in transit. The result (f) is a not a BFS spanning tree.

xP Specification 3.7: SynchBFS

Input: All cells start in the same initial state, S_2, with the same set of rules. Each cell, σ_i, contains an immutable cell ID symbol, ι_i, and neighbour pointers, n_j's. The source cell, σ_s, is additionally marked with one symbol, s.

Output: All cells end in the same state, S_2; neighbour pointer symbols and cell IDs are intact. Cell σ_s is still marked with one s. Each cell contains a visited mark, v, and a spanning tree parent pointer, p_j. Specifically, the source cell, σ_s, contains one p_s indicating it is the root of the spanning tree.

Symbols and states

Cell σ_i uses the following symbols and states.

- f_i is its visit token;

- n_j indicates its neighbour, σ_j;
- p_j indicates its spanning tree parent, σ_j;
- f indicates that it is a frontier cell;
- v indicates that it is visited.

State S_2 is used as both unvisited and visited states.

The matrix R of xP Specification 3.7 consists of only one vector.

1. Main search

1.1. $S_2 \ \rightarrow_{\texttt{min.min}} \ S_2 \ f_i \mid \iota_i \ s \ \neg \ v$

1.2. $S_2 \ f_j \ \rightarrow_{\texttt{min.min}} \ S_2 \ f \ v \ p_j \ \neg \ v$

1.3. $S_2 \ \rightarrow_{\texttt{max.min}} \ S_2 \ (f_i) \updownarrow_k \ \mid \iota_i \ f \ n_k \ \neg \ p_k$

1.4. $S_2 \ f \ \rightarrow_{\texttt{min}} \ S_2$

1.5. $S_2 \ f_j \ \rightarrow_{\texttt{max.max}} \ S_2 \mid v$

Initial and final configurations

Table 3.14 shows the initial and final configurations of xP Specification 3.7 for Figure 3.11 (synchronous mode).

Table 3.14: Initial and final configurations of xP Specification 3.7 for Figure 3.11 (synchronous mode).

Time	σ_1	σ_2	σ_3	σ_4
0	$S_2 \ s \ \iota_1 \ n_2 \ n_3 \ n_4$	$S_2 \ \iota_2 \ n_1 \ n_3 \ n_4$	$S_2 \ \iota_3 \ n_1 \ n_2 \ n_4$	$S_2 \ \iota_4 \ n_1 \ n_2 \ n_3$
2	$S_2 \ s \ \iota_1 \ n_2 \ n_3 \ n_4 \ v \ \mathbf{p_1}$	$S_2 \ \iota_2 \ n_1 \ n_3 \ n_4 \ v \ \mathbf{p_1}$	$S_2 \ \iota_3 \ n_1 \ n_2 \ n_4 \ v \ \mathbf{p_1}$	$S_2 \ \iota_4 \ n_1 \ n_2 \ n_3 \ v \ \mathbf{p_1}$

Rule explanations

The source cell, σ_s, generates one token, f_s, which simulates that σ_s receives a visit token from a non-existing cell (rule 1.1). Cell σ_i handles the received tokens, f_j, using the following rules corresponding to the token handling rules discussed in Section 3.4.1.

(I) Rule 1.2: when unvisited cell σ_i, $\neg \ v$, receives f_j's, it selects one of the sending cells, by `min.min` mode, as its parent, p_j, marks itself as visited by v and generates one f, indicating it is a frontier cell.

(a) Rules 1.3–4: as a frontier, σ_i sends f_i to all non-parent neighbours, $n_k \ \neg \ p_k$.

(II) Rule 1.5: visited cell σ_i, v, deletes all received f_j's.

Partial traces

Table 3.15 shows the partial traces of xP Specification 3.7 for cell σ_3 in Figure 3.11 (synchronous mode). Omitted symbols (...) are ι_3 n_1 n_2 n_4 (see Table 3.14).

Table 3.15: Partial traces of xP Specification 3.7 for cell σ_3 in Figure 3.11 (synchronous mode).

Fig.	Evolution	Content
(a)		...
(b)	$\{f_1\} \Rightarrow v\ p_1\ \{f_3\}_{2,4}$	$v\ p_1 \ldots$
(c)	$\{f_2\ f_4\} \Rightarrow$	$v\ p_1 \ldots$

3.4.3 Asynchronous BFS (AsynchBFS)

Asynchronous BFS (AsynchBFS) algorithm [42] supplements SynchBFS with rules that correct the parent designation and *guarantees* a BFS spanning tree (instead of an arbitrary spanning tree) in the *asynchronous* mode [3].

Initially, the source cell, σ_s, sets its distance, $d_s = 0$. Each cell, σ_i, records its *distance*, l, from the source cell, measured in *search path lengths*, denoted as $d_i = l$. A cell, σ_i, sends a *distance notification* of its distance l,

1. together with its visit token: the visit token together with a distance notification is called a *v-token*, denoted as $v(u_i = l)$; or
2. when it updates its distance: the distance notification is denoted as $u_i = l$.

The above notations are summarised as below:

distance:	$d_i = l$
v-token (visit token + distance notification):	$v(u_i = l)$
distance notification:	$u_i = l$

Consider a cell, σ_i, which receives

- one or more v-tokens (sent in case 1), and/or
- one or more distance notifications (sent in case 2).

Cell σ_i computes the minimum of the received distances, u_j: (1) if σ_i has not recorded its distance, it sets $d_i = u_j + 1$; (2) otherwise, σ_i compares d_i with $u_j + 1$, if $u_j + 1 < d_i$, it sets $d_i = u_j + 1$ and changes its parent to σ_j.

If cell σ_i sets or updates its distance, it sends its distances notification, u_i, to all its non-parent neighbours.

Example 3.11. Figure 3.13 shows one snippet of the evolution of the AsynchBFS algorithm in one possible asynchronous run.

(a) The source cell, σ_1, initialises its distance, $d_1 = 0$, and broadcasts its v-token, $v(u_1 = 0)$.

(b) Unvisited cell σ_2 receives $v(u_1 = 0)$ from σ_1, so it marks itself as visited, sets its parent as σ_1 and $d_2 = 1$, and sends $v(u_2 = 1)$ to σ_3 and σ_4.

(c) Unvisited cell σ_3 receives $v(u_2 = 1)$ from σ_2, so it marks itself as visited, sets its parent as σ_2 and $d_3 = 2$, and sends $v(u_3 = 2)$ to σ_1 and σ_4.

(d) Unvisited cell σ_4 receives $v(u_3 = 2)$ from σ_3, so it marks itself as visited, sets its parent as σ_3 and $d_4 = 3$, and sends $v(u_4 = 3)$ to σ_1 and σ_2.

Visited cell σ_1 receives $v(u_3 = 2)$ from σ_3, but it finds that $u_3+1 > d_1$, therefore ignoring this v-token.

(e) Visited cell σ_1 receives $v(u_4 = 3)$ from σ_4, but it finds that $u_4+1 > d_1$, therefore ignoring this v-token.

Visited cell σ_2 receives $v(u_4 = 3)$ from σ_4, but it finds that $u_4+1 > d_2$, therefore ignoring this v-token.

Assume that v-tokens $v(u_1 = 0)$ from σ_1 to σ_3, $v(u_1 = 0)$ from σ_1 to σ_4 and $v(u_2 = 1)$ from σ_2 to σ_4 arrive.

(f) Visited cell σ_3 finds that $u_1 + 1 < d_3$, so it updates $d_3 = 1$, changes its parent from σ_2 to σ_1, and sends $u_3 = 1$ to σ_2 and σ_4.

Visited cell σ_4 computes the minimum of received distances, $min(u_1, u_2) = u_1 = 0$. Because $u_1 + 1 < d_4$, σ_4 updates $d_4 = 1$, changes its parent from σ_3 to σ_1, and sends $u_4 = 1$ to σ_2 and σ_3.

(g) On receiving $u_3 = 1$ and $u_4 = 1$, visited cell σ_2 computes the minimum of received distances, $min(u_3, u_4) = u_3(= u_4) = 1$. Because $u_3 + 1 > d_2$, σ_2 simply ignores it.

On receiving $u_4 = 1$, visited cell σ_3 finds $u_4 + 1 > d_3$, so it simply ignores it.

On receiving $u_3 = 1$, visited cell σ_4 finds $u_3 + 1 > d_4$, so it simply ignores it.

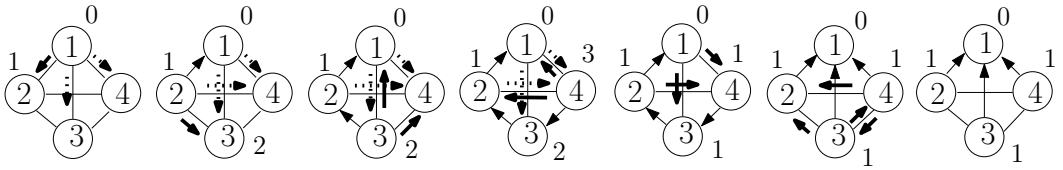


Figure 3.13: An example of AsynchBFS in one possible asynchronous run. Arcs with arrows: virtual spanning tree child-parent arcs; thick arrows near arcs: visit tokens; dotted arrows: messages still in transit; the number beside each cell: the cell's distance.

In AsynchBFS, no cell knows when the algorithm terminates (this would be true even if the size of network were known) [42]. To solve this problem, we can apply a termination algorithm to this algorithm, as later discussed in Section 3.6.

xP Specification 3.8: AsynchBFS

Input: Same as in xP Specification 3.7.

Output: All cells end in the same state, S_2; neighbour pointer symbols and cell IDs are intact. Cell σ_s is still marked with one s. Each cell contains a visited mark, v, a distance indicator, $d_i(c^l)$, and a BFS spanning tree parent pointer, p_j. Specifically, the source cell, σ_s, contains one p_s, indicating it is the root of the BFS spanning tree.

Symbols and states

Our xP Specification 3.8 of AsynchBFS uses the same symbols and state as xP Specification 3.7 of SynchBFS, plus a few more specific. Cell σ_i uses the following additional symbols:

- $d_i(c^l)$ records its own distance, l;
- $u_i(c^l)$ is its distance notification;
- $f_i\ u_i(c^l)$ is a v-token consisting of a visit token symbol, f_i and a distance notification symbol, $u_i(c^l)$;
- $m_j(c^l)$ records σ_j's notified distance plus one;
- $m'_j(c^l)$ is the intermediate or final computed minimum result of all received distances plus one;
- m indicates that it must next send its distance notification.

The matrix R of xP Specification 3.8 consists of only one vector.

1. Main search

1.1. $S_2 \ \rightarrow_{\texttt{min.min}} \ S_2 \ f_i \ u_i(\lambda) \mid \iota_i \ s \ \neg \ v$

1.2. $S_2 \ f_j \ \rightarrow_{\texttt{min.min}} \ S_2 \ f \ v \ p_j \mid \iota_i \ \neg \ v$

1.3. $S_2 \ \rightarrow_{\texttt{max.min}} \ S_2 \ (f_i) \updownarrow_j \ \mid \iota_i \ f \ n_j \ \neg \ p_j$

1.4. $S_2 \ f \ \rightarrow_{\texttt{min}} \ S_2$

1.5. $S_2 \ f_j \ \rightarrow_{\texttt{max.max}} \ S_2 \mid v$

1.6. $S_2 \ u_j(X) \ \rightarrow_{\texttt{max.min}} \ S_2 \ m_j(Xc)$

1.7. $S_2 \ \rightarrow_{\texttt{min.min}} \ S_1 \ m'_j(X) \mid m_j(X)$

1.8. $S_2 \ m'_j(XY) \ \rightarrow_{\texttt{max.min}} \ S_1 \ m'_j(X) \mid m_j(X)$

1.9. $S_2 \ m'_j(XY) \ \rightarrow_{\texttt{min.min}} \ S_2 \mid \iota_i \ d_i(X)$

1.10. $S_2 \ d_i(XY) \ \rightarrow_{\texttt{min.min}} \ S_2 \mid \iota_i \ m'_j(X)$

1.11. $S_2 \ m'_j(X) \ p_k \ \rightarrow_{\texttt{min.min}} \ S_2 \ d_i(X) \ p_j \ m \mid \iota_i$

1.12. $S_2 \ \rightarrow_{\texttt{max.min}} \ S_2 \ (u_i(X)) \updownarrow_j \ \mid \iota_i \ m \ n_j \ d_i(X) \ \neg \ p_j$

1.13. $S_2 \ m_j(X) \ \rightarrow_{\texttt{max.max}} \ S_2$

1.14. $S_2 \ m \ \rightarrow_{\texttt{min}} \ S_2$

Initial and final configurations

Table 3.16 shows initial and final configurations of xP Specification 3.8 for Figure 3.13, highlighting the BFS spanning tree (there is only one BFS tree in this case).

Table 3.16: Initial and final configurations of xP Specification 3.8 for Figure 3.13.

Time	σ_1	σ_2	σ_3	σ_4
0	$S_2 \ s \ \iota_1 \ n_2 \ n_3 \ n_4$	$S_2 \ \iota_2 \ n_1 \ n_3 \ n_4$	$S_2 \ \iota_3 \ n_1 \ n_2 \ n_4$	$S_2 \ \iota_4 \ n_1 \ n_2 \ n_3$
2	$S_2 \ s \ \iota_1 \ n_2 \ n_3 \ n_4 \ v \ \mathbf{p_1}$ $d_1(c)$	$S_2 \ \iota_2 \ n_1 \ n_3 \ n_4 \ v \ \mathbf{p_1}$ $d_2(c^2)$	$S_2 \ \iota_3 \ n_1 \ n_2 \ n_4 \ v \ \mathbf{p_1}$ $d_3(c^2)$	$S_2 \ \iota_4 \ n_1 \ n_2 \ n_3 \ v \ \mathbf{p_1}$ $d_4(c^2)$

Rule explanations

The source cell, σ_s, generates one v-token, f_s and $u_s(\lambda)$, which simulates that σ_s receives a v-token from a non-existing cell (rule 1.1); therefore, σ_s's distance is set to 1 (instead of 0 as mentioned before).

Cell σ_i handles its received visit token, using the following rules corresponding to the token handling rules discussed in Section 3.4.1:

(I) Rule 1.2: when unvisited cell σ_i, v, receives v-tokens, f_j's and $u_j(X)$'s, it selects one of the sending cells, σ_j, using `min.min` mode, as its parent, p_j, marks itself as visited by v and generates one f, indicating it is a frontier cell; also, σ_i computes its distance and sends its distance notification as later discussed.

(a) Rules 1.3–4: as a frontier, σ_i sends f_i to all its non-parent neighbours, $n_j \neg p_j$.

(II) Rule 1.5: when visited cell σ_i, v, receives v-tokens, f_j's and $u_j(X)$'s, it deletes all f_j's and computes its distance as below.

The following rules are used to set or update a cell's distance. Consider cell σ_i, which receives one or more distance notifications, $u_j(X)$'s, (sent together with a visit token as in (a) or on its own when it updates its distance as discussed below). Cell σ_i computes the minimum of the received distances plus one, $m'_j(Z')$ (rules 1.6–8).

- If it has no $d_i(Z)$, it directly sets its distance as $m'_j(Z')$ (rules 1.9–10 skipped) and generates one m (rule 1.11, in this case, $j = k$), indicating that it must next send its distance notification (rule 1.12).
- Otherwise, it compares $d_i(Z)$ with $m'_j(Z')$:
  - if the multiplicity of c represented by Z <= the multiplicity of c represented by Z', it removes $m'_j(Z')$ (rule 1.9);
  - otherwise, it erases its old distance, $d_i(Z)$ (rule 1.10), records its new distance by $d_i(Z')$, sets its parent as p_j and generates one m (rule 1.11, in this case, $j \neq k$), indicating that it must next send its distance notification (rule 1.12);

If cell σ_i has one m, it sends its distance notification, $u_i(Z')$, to all its non-parent neighbours, $n_j \neg p_j$, and deletes m (rules 1.12, 1.14).

Partial traces

Table 3.17 shows partial traces of xP Specification 3.8 of xP Specification 3.8 for cell σ_3 in Figure 3.13. Omitted symbols (...) are $\iota_3\ n_1\ n_2\ n_4$ (see Table 3.16).

Table 3.17: Partial traces of xP Specification 3.8 for cell σ_3 in Figure 3.13.

Fig.	Evolution	Content
(a)		...
(b)		...
(c)	$\{f_2\ u_2(c^2)\} \Rightarrow v\ p_2\ d_3(c^3)\ \{f_3\ u_3(c^3)\}_{1,4}$	$v\ p_2\ d_3(c^3)\ldots$
(d)		$v\ p_2\ d_3(c^3)\ldots$
(e)		$v\ p_2\ d_3(c^3)\ldots$
(f)	$\{f_1\ u_1(c)\}\ p_2\ d_3(c^3) \Rightarrow p_1\ d_3(c^2)\ \{u_3(c^2)\}_{2,4}$	$v\ p_1\ d_3(c^2)\ldots$
(g)	$\{u_4(c^2)\} \Rightarrow$	$v\ p_1\ d_3(c^2)\ldots$

3.5 Neighbour Discovery in Undirected Graphs

Echo, distributed DFS and BFS algorithms assume that each cell knows its neighbours. Following our joint work [3], this section removes this assumption by running a preliminary phase before one of these traversal algorithms, which builds the neighbourhood knowledge in each cell. We first provides a synchronous neighbour discovery (SynchND) algorithm, and then presents an asynchronous neighbour discovery (AsynchND) algorithm.

Both SynchND and AsynchND are broadcast-based algorithms, which do not require any topology knowledge in cells, not even the number of their neighbours.

3.5.1 Synchronous Neighbour Discovery (SynchND)

The source starts to broadcast its visit token, which encodes its cell ID. The token handling rules are shown as follows.

(I) When an *unvisited* cell receives visit tokens, it marks itself as visited and broadcasts its visit token, which encodes its own cell ID.

(II) A *visited* cell just receive visit tokens.

(III) Two time units after being visited, a cell *knows* that it has received all its neighbours' visit tokens.

The algorithm terminates when no more tokens are sent, but no cell knows when the algorithm terminates. This is not required if a subsequent phase starts two time units after this algorithm. This delay guarantees that each cell will have had enough time to compute its neighbourhood.

However, in the asynchronous mode, the message transmission delay is arbitrary and thus a cell does not know when it receives all its neighbours' visit tokens. An asynchronous solution is later presented in Section 3.5.2.

Example 3.12. Figure 3.14 shows the evolution of SynchND algorithm in the *synchronous* mode.

(a) At the start, the source cell, σ_1, broadcasts its visit token.

(b) On receiving σ_1's token, unvisited cells σ_2, σ_3 and σ_4 broadcast visit tokens.

(c) Visited cells σ_2, σ_3 and σ_4 receive all their neighbours' visit tokens.

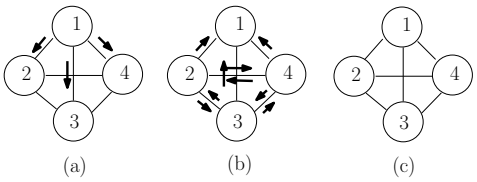


Figure 3.14: An example of SynchND in the synchronous mode. Thick arrows near edges indicate visit tokens.

xP Specification 3.9: Synchronous Neighbour Discovery (SynchND)

Input: All cells start in the same state, S_0, with the same set of rules. Each cell, σ_i, contains an immutable cell ID symbol, ι_i. The source cell, σ_s, is additionally marked with one symbol, s.

Output: All cells end in the same state, S_2, and cell IDs are intact. Cell σ_s is still marked with one s. Each cell contains its neighbour pointers, n_j's.

Symbols and states

Cell σ_i sends out n_i as its visit token. State S_0 is an unvisited state; states S_1 and S_2 are visited states. The transition from S_0 to S_1 and then to S_2 implements two time units for each cell to receive all its neighbours' visit tokens.

The matrix R of xP specification 3.9 consists of two vectors, informally presented in two groups, according to their functionality and applicability.

1. An unvisited cell receives visit tokens.

1.1. $S_0 \ \rightarrow_{\texttt{min.min}} \ S_1 \ (n_i) \updownarrow \ | \ \iota_i \ s$

1.2. $S_0 \ \rightarrow_{\texttt{min.min}} \ S_1 \ (n_i) \updownarrow \ | \ \iota_i \ n_j$

2. A visited cell receives visit tokens automatically.

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

Initial and final configurations

Table 3.18 shows the initial and final configurations of xP Specification 3.9 for Figure 3.14.

Table 3.18: Initial and final configurations of xP Specification 3.9 for Figure 3.14.

Time	σ_1	σ_2	σ_3	σ_4
0	$S_0 \ s \ \iota_1$	$S_0 \ \iota_2$	$S_0 \ \iota_3$	$S_0 \ \iota_4$
3	$S_2 \ s \ \iota_1 \ \mathbf{n_2 \ n_3 \ n_4}$	$S_2 \ \iota_2 \ \mathbf{n_1 \ n_3 \ n_4}$	$S_2 \ \iota_3 \ \mathbf{n_1 \ n_2 \ n_4}$	$S_2 \ \iota_4 \ \mathbf{n_1 \ n_2 \ n_3}$

Rule explanations

The source cell, σ_s, starts to broadcast its visit token, n_s, to all its neighbours (rule 1.1). Cell σ_i handles the received visit tokens, n_j's, using the following rules corresponding to the token handling rules of SynchND.

(I) Rule 1.2: when unvisited cell σ_i (in state S_0) receives n_j's, it broadcasts its visit token and marks itself as visited by entering state S_1.

(II) This is automatically done and no rule is needed.

(III) Rule 2.1: a cell receives all its neighbours' visit tokens in state S_2 (two time units).

3.5.2 Asynchronous Neighbour Discovery (AsynchND)

In AsynchND, a cell does not wait two time units to receive all visit tokens (because a cell does not know when it receives all visit tokens in the asynchronous mode). Similarly, the source starts to broadcast its visit token, which encodes its cell ID. The token handling rules are the same as SynchND, except that rule (III) of SynchND is not used.

When this algorithm terminates, no cells are aware of it. This problem can be solved by applying a termination detection algorithm (as later discussed in Section 3.6), so that the source cell knows the algorithm termination.

Figure 3.15 shows how AsynchND works in one possible asynchronous run.

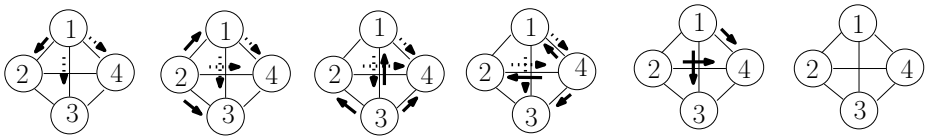


Figure 3.15: An example of AsynchND in one possible asynchronous run. Thick arrows near edges: visit tokens; dotted arrows: messages still in transit.

xP Specification 3.10: Asynchronous Neighbour Discovery (AsynchND)

Input: Same as in xP Specification 3.9, except that all cells start in state S_1.

Output: Same as in xP Specification 3.9.

Symbols and states

Cell σ_i sends out n_i as its visit token. State S_1 is an unvisited state and state S_2 is a visited state: two states are used instead of three states as in xP Specification 3.9 of SynchND.

The matrix R of xP specification 3.9 consists of only one vector.

1. An unvisited cell receives visit tokens.

1.1. $S_1 \ \rightarrow_{\texttt{min.min}} \ S_2 \ (n_i) \updownarrow \ | \ \iota_i \ s$

1.2. $S_1 \ \rightarrow_{\texttt{min.min}} \ S_2 \ (n_i) \updownarrow \ | \ \iota_i \ n_j$

Initial and final configurations

Table 3.19 shows the initial and final configurations of xP Specification 3.10 for Figure 3.15.

Table 3.19: Initial and final configurations of xP Specification 3.10 for Figure 3.15.

Time	σ_1	σ_2	σ_3	σ_4
0	$S_1 \ s \ \iota_1$	$S_1 \ \iota_2$	$S_1 \ \iota_3$	$S_1 \ \iota_4$
1	$S_2 \ s \ \iota_1 \ \mathbf{n_2 \ n_3 \ n_4}$	$S_2 \ \iota_2 \ \mathbf{n_1 \ n_3 \ n_4}$	$S_2 \ \iota_3 \ \mathbf{n_1 \ n_2 \ n_4}$	$S_2 \ \iota_4 \ \mathbf{n_1 \ n_2 \ n_3}$

Rule explanations

The rule explanations are the same as in xP Specification 3.9 of SynchND, except that rule 2.1 of SynchND is not used: a cell receives all its neighbours' visit tokens in visited state S_2 (but the cell does not know when it receives all visit tokens).

3.6 Termination Detection and Announcement

As discussed in Section 1.1, termination detection and announcement aim to convert an implicitly terminating distributed algorithm, A, into an equivalent explicitly terminating algorithm, A', which can be achieved by using two algorithms [62]:

1. a *termination detection* algorithm that converts A into a partial-explicit terminating algorithm (§ 1.1), A^*, in which the source cell knows but not all cells know termination;

2. a *termination announcement* algorithm that converts A^* into an explicit terminating algorithm (§ 1.1), A', in which all cells know termination.

Termination announcement [62] is simple (it can be done by a broadcast phase) and is not always necessary if not all processes need to know termination. This section studies termination detection.

There are several well defined ways to detect the termination of algorithm A. Intuitively, this can be clearly detected from outside, by an external powerful observer,

who can continuously probe all cells and all communication channels (without interfering with their current processes). However, can the cells themselves detect this termination? For this purpose, cells can run a termination detection algorithm, B, ideally as a *control layer* over algorithm A.

In order to differentiate algorithm A with termination detection algorithm B, A is called the *basic algorithm* while B is called the *control algorithm*; messages in A are *basic messages* while messages in B are *control messages*. Control algorithm B runs in parallel with basic algorithm A, interacting with A at specific points. To make a clean separation, in xP systems, we describe this combination as a *parallel composition with interaction* (§ 2.3.7).

The maximum delay between A's termination and its detection by B is *detection latency*, which is the time complexity of the termination detection algorithm, B.

We present xP solutions of several well-known termination detection algorithms:

- *synchronous* ring-based Dijkstra-Feijen-Van Gasteren (DFG) algorithm,
- *asynchronous* ring-based Safra's algorithm and
- *asynchronous* computation-tree-based Dijkstra-Scholten algorithm.

The basic algorithm and the termination detection algorithm use the *same single source* cell. By applying a termination detection algorithm to a basic algorithm, we obtain a partial-explicit terminating traversal algorithm, where the source cell knows when the basic algorithm terminates, but other cells are not aware of the termination:

- SynchBFS+DFG, which applies the DFG algorithm to SynchBFS;
- AsynchBFS+Safra, which applies Safra's algorithm to AsynchBFS;
- AsynchBFS+DS, which applies the Dijkstra-Scholten algorithm to AsynchBFS;
- AynchND+DS, which applies the Dijkstra-Scholten algorithm to AsynchND;
- SynchBFS+DS, which applies the Dijkstra-Scholten algorithm to SynchBFS.

With λ messaging (§ 2.3.1), all xP systems conform to usual activation assumptions (§ 1.1). Specifically, for all basic algorithms used in this section, SynchBFS (§ 3.4.2), AsynchBFS (§ 3.4.3) and AsynchND (§ 3.5), no rule application make any rule applicable at the start of the next step without receiving a message. Thus, no λ message is sent in all these basic algorithms and we do not need to consider λ messages in the termination detection.

Each partial-explicit terminating xP solution, $\Pi_1^* \triangleright \Pi_2$, is constructed by parallel composition with interaction (§ 2.3.7).

- Π_1^*, a *modified* version of the basic algorithm, Π_1, which "feeds" symbols to the termination detection algorithm;
- Π_2, the *control layer* of the termination detection algorithm.

3.6.1 The Dijkstra-Feijen-Van Gasteren Algorithm

The Dijkstra-Feijen-Van Gasteren (DFG) algorithm assumes *synchronous* messaging and an underlying network containing a Hamiltonian cycle (also called a *ring*). It checks whether all cells are passive by passing a *DFG token*, called a *d-token*, around the ring using a black and white colouring scheme.

This algorithm is *round-based*, which detects termination by *repeated* rounds: each round starts when the source cell sends a d-token and ends when the source cell receives back the d-token and becomes passive.

Assume that the basic algorithm is extended with ingredients for the DFG algorithm: a *white or black* property for each cell and a global *white or black d-token*, the DFG algorithm detects termination of the basic algorithm based on the following *detection rules*.

Additions to the basic algorithm:

(I) Initially, all cells are white.

(II) A (source or non-source) cell that sends a basic message becomes black.

(III) When the source cell becomes *passive* in the basic algorithm, it sends a white d-token to start the first round (this is done only once).

Control layer of the DFG algorithm:

(IV) A non-source cell only forwards the d-token when it is *passive* in the basic algorithm.

(V) When a black non-source cell forwards the d-token, the d-token becomes black (if it is white).

(VI) Each (source or non-source) cell becomes white (if it is black) immediately after forwarding the d-token.

(VII) When the d-token returns to the source cell, the source cell waits until it is *passive* in the basic algorithm:

(a) if the d-token and the source cell are white, the source cell knows termination;

(b) otherwise, the source cell sends a white d-token again to start another round.

Rules (IV) and (VII) of the DFG algorithm can be only applied when a cell would become *passive* (§ 1.2) in the basic algorithm. In xP systems, this is achieved by the parallel composition with interaction (§ 2.3.7). The rules of the DFG algorithm have lower priority than the rules of the basic algorithm and thus can be only applied when a cell can not apply any more rules of the basic algorithm, i.e. when a cell would become passive in the basic algorithm.

As a simple illustration, Example 3.13 shows the most straightforward way to detect termination for SynchBFS. If known, a timer of diam in the source cell, where diam is the diameter of the graph, is adequate to detect termination, because SynchBFS runs in diam time units.

Another way to detect termination for SynchBFS is using a convergecast phase to inform the source cell of the termination in diam time units, which is a special case of SynchBFS+DS, as later discussed in Section 3.6.3.

Example 3.13. Figure 3.16 shows how to apply the DFG algorithm to SynchBFS in the *synchronous* scenario in Section 3.4.2. Graph G contains a ring, $\sigma_1.\sigma_2.\sigma_3.\sigma_4.\sigma_1$.

(a) At the start, the source cell, σ_1, broadcasts its visit token and becomes black.

(b) The source cell, σ_1, sends a white d-token to σ_2 to start round 1. Cells σ_2, σ_3 and σ_4 send visit tokens and become black.

(c) On receiving the white d-token, black σ_2 sends a black d-token to σ_3 and becomes white.

(d) On receiving the black d-token, black σ_3 sends a black d-token to σ_4 and becomes white.

(e) On receiving the black d-token, black σ_4 sends a black d-token to σ_1 and becomes white.

(f) The black source cell, σ_1, receives back the black d-token, so it sends a white d-token again to start round 2.

(g)–(i) The white d-token travels around the ring.

(j) Finally, the white source cell, σ_1, receives back a white d-token and thus knows termination.

The correctness and liveness theorem in Tel's book [62] indicate that the detection latency of the DFG algorithm is *one or two rounds*. Consider round r when the basic algorithm terminates.

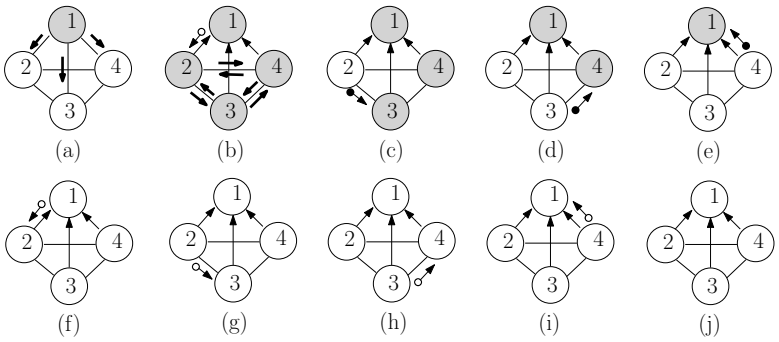


Figure 3.16: An example of SynchBFS+DFG in the synchronous mode. Edges with arrows: virtual spanning tree child-parent arcs; thick arrows near edges: basic messages (visit tokens); thin arrows with white or black circles near edges: white or black d-tokens respectively.

(1) If the d-token makes all cells white in round r, then round $r+1$ detects termination.

(2) Otherwise (for the cells that becomes black after the d-token passes by, the d-token can not make them white in round r), the d-token makes all cells white in round $r+1$ and round $r+2$ detects termination.

Thus, the runtime complexity of the DFG algorithm is $O(n)$. Example 3.13 shows case (1), in which the basic algorithm terminates in round 1 and round 2 detects termination.

However, in the asynchronous mode, the DFG algorithm can not guarantee to detect termination: even though a d-token finds all cells passive, there may be messages still in transit, which can make cells active again. This problem is solved by Safra's algorithm in Section 3.6.2.

xP Specification 3.11: Control layer of the DFG algorithm

Input: Assumptions of the basic algorithm are made: (I) initially, all cells contain no b'; (II) a cell that sends a basic message generates one b' if it does not contain b'; (III) the source cell, σ_s, sends a loopback message of one t when it starts computation. Additionally, each cell, σ_i, contains its ring successor pointer, r_k.

Output: All ring successor pointer symbols are intact. The source cell, σ_s, contains one g, indicating it knows the algorithm termination.

Symbols and states

Cell σ_i uses the following symbols:

- r_j indicates a ring successor, σ_j;

- w is a white d-token;
- b is a black d-token;
- b' indicates that it is black (white if it does not contain b');
- t indicates that it must next send a black or white d-token;
- g indicates that it knows the algorithm termination.

The control layer of the DFG algorithm uses only one state, S_1.

The matrix R of xP Specification 3.11 consists of only one vector.

1. Detection

1.1. $S_1\ w\ \rightarrow_{\texttt{min}}\ S_1\ g \mid s \neg b'$

1.2. $S_1\ w\ \rightarrow_{\texttt{min}}\ S_1\ t$

1.3. $S_1\ b\ \rightarrow_{\texttt{min}}\ S_1\ t$

1.4. $S_1\ t\ b'\ \rightarrow_{\texttt{min.min}}\ S_1\ (w) \updownarrow_j \mid s\ r_j$

1.5. $S_1\ t\ b'\ \rightarrow_{\texttt{min.min}}\ S_1\ (b) \updownarrow_j \mid r_j$

1.6. $S_1\ t\ \rightarrow_{\texttt{min.min}}\ S_1\ (w) \updownarrow_j \mid r_j \neg b'$

Rule explanations

The xP rules correspond to the detection rules of the control layer of the DFG algorithm.

(IV) Rules 1.2–3: when cell σ_i receives a white or black d-token, w or b, it generates one t, indicating it must next send a d-token (rules 1.2–3).

Then if σ_i is white, $\neg\ b'$, it sends a white d-token, w, to its ring successor, r_j, and deletes t (rule 1.6).

(V) Rule 1.5: otherwise (if σ_i is black, b'), it sends a black d-token, b, to its ring successor, r_j, becomes white by erasing b' and deletes t.

(VI) Rule 1.5: as discussed in (V).

(VII) Rules 1.1–4, 1.6: consider the source cell, σ_s, which receives w or b.

(a) Rule 1.1: if σ_s, receives w and is white, $\neg\ b'$, then it generates one g, indicating that it knows termination.

(b) Rules 1.2–4, 1.6: otherwise, it generates one t (rules 1.2–3); then it sends w to start another round and deletes t (rules 1.4, 1.6). Also, if σ_s is black, b', it becomes white by erasing b' (rule 1.4).

Algorithm SynchBFS+DFG

SynchBFS+DFG solves the termination detection problem of SynchBFS by applying the DFS algorithm. Example 3.13 illustrates the evolution of SynchBFS+DFG in the synchronous mode.

xP Specification 3.12: SynchBFS+DFG

Input: Same as in xP Specification 3.7. Additionally, each cell, σ_i, contains its ring successor pointer, r_k.

Output: Same as in xP Specification 3.7. All ring successor pointer symbols are intact. Additionally, the source cell, σ_s, contains one g, indicating it knows the algorithm termination.

Symbols and states

The modified SynchBFS, Π_1^*, uses the same symbols as xP Specification 3.7 of SynchBFS, plus a few more specific to the DFG algorithm. Cell σ_i uses the following additional symbols:

- r_j indicates a ring successor, σ_j;
- b' indicates that it is black (white if it does not contain b');
- t indicates that it must next send a black or white d-token.

The matrix R of xP Specification 3.12 consists of only one vector.

1. Main search and detection

$\overline{\Pi_1^*}$: rules of the modified version of SynchBFS by replacing S_2 with $\Theta(S_2, Y)$, where boxed rules correspond to additions to SynchBFS for the DFG algorithm.

1.1. $\boxed{\Theta(S_2,Y) \ \rightarrow_{\texttt{min.min}} \ \Theta(S_2,Y)\ (t)\ f_i \mid \iota_i\ s \neg v}$

1.2. $\Theta(S_2,Y)\ f_j \ \rightarrow_{\texttt{min.min}} \ \Theta(S_2,Y)\ f\ v\ p_j \neg v$

1.3. $\boxed{\Theta(S_2,Y) \ \rightarrow_{\texttt{min}} \ \Theta(S_2,Y)\ b' \mid f \neg b'}$

1.4. $\Theta(S_2,Y) \ \rightarrow_{\texttt{max.min}} \ \Theta(S_2,Y)\ (f_i) \updownarrow_j \ \mid \iota_i\ f\ n_j \neg p_j$

1.5. $\Theta(S_2,Y)\ f \ \rightarrow_{\texttt{min}} \ \Theta(S_2,Y)$

1.6. $\Theta(S_2,Y)\ f_j \ \rightarrow_{\texttt{max.max}} \ \Theta(S_2,Y) \mid v$

$\overline{\Pi_2}$: rules of the control layer of the DFG algorithm by replacing S_1 with $\Theta(X, S_1')$.

Table 3.20: Initial and final configurations of xP Specification 3.12 for Figure 3.16.

Time	σ_1	σ_2	σ_3	σ_4
0	$\Theta(S_2, S_1')$ s ι_1 r_2 n_2 n_3 n_4	$\Theta(S_2, S_1')$ ι_2 r_3 n_1 n_3 n_4	$\Theta(S_2, S_1')$ ι_3 r_4 n_1 n_2 n_4	$\Theta(S_2, S_1')$ ι_4 r_1 n_1 n_2 n_3
10	$\Theta(S_2, S_1')$ s ι_1 r_2 v p_1 n_2 n_3 n_4 $\boldsymbol{g}$	$\Theta(S_2, S_1')$ ι_2 r_3 v p_1 n_1 n_3 n_4	$\Theta(S_2, S_1')$ ι_3 r_4 v p_1 n_1 n_2 n_4	$\Theta(S_2, S_1')$ ι_4 r_1 v p_1 n_1 n_2 n_3

Initial and final configurations

Table 3.20 shows initial and final configurations of xP Specification 3.12 for Figure 3.16.

Rule explanations

In SynchBFS, a message is a visit token, f_j. The xP rules of the modified SynchBFS correspond to the detection rules of additions to the basic algorithm for the DFG algorithm.

(I) Initially, all cells contain no b'.

(II) Rule 1.3: symbol f indicates that a cell must next send visit tokens, so rule 1.3 is added, which uses f as a promoter to generate b' if no b' exists.

(III) Rule 1.1: the source cell, σ_s, sends a loopback message of one t, indicating that it must next send a d-token.

Partial traces

Table 3.21 shows partial traces of xP Specification 3.12 for cell σ_3 in Figure 3.16, highlighting the symbols used for the DFG algorithm. Omitted symbols (...) are ι_3 n_1 n_2 n_4 (see Table 3.20).

Table 3.21: Partial traces of xP Specification 3.12 for cell σ_3 in Figure 3.11 (synchronous mode).

Fig.	Evolution	Content
(a)		$r_4 \ldots$
(b)	$\{f_1\} \Rightarrow v\ p_1\ \boldsymbol{b'}\ \{f_3\}_{2,4}$	$r_4\ v\ p_1\ \boldsymbol{b'} \ldots$
(c)	$\{f_2\ f_4\} \Rightarrow$	$r_4\ v\ p_1\ \boldsymbol{b'} \ldots$
(d)	$\{\boldsymbol{b}\}\ \boldsymbol{b'} \Rightarrow \{\boldsymbol{b}\}_4$	$r_4\ v\ p_1 \ldots$
(e) (f) (g)		$r_4\ v\ p_1 \ldots$
(h)	$\{\boldsymbol{w}\} \Rightarrow \{\boldsymbol{w}\}_4$	$r_4\ v\ p_1 \ldots$
(i)		$r_4\ v\ p_1 \ldots$

3.6.2 Safra's Algorithm

Safra's algorithm generalises the DFG algorithm in the *asynchronous* setting, which is also *round-based*. Each cell, σ_i, maintains a signed integer *s-counter*, $\sigma_i.c$, which records the number of basic messages it sent minus the number of basic messages it received; the global *s-token* carries *s-sum*, which records the sum of the values of s-counters it traversed. In this way, the source cell can compare the number of basic messages that are sent and received in the system to ensure all channels are empty.

Assume that the basic algorithm is extended with ingredients for Safra's algorithm: a white or black property and an s-counter for each cell and a global white or black s-token with the s-sum, Safra's algorithm detects termination of the basic algorithm based on the following detection rules.

Additions to the basic algorithm:

(I) Initially, each cell is white and its s-counter is set to zero, $\sigma_i.c = 0$.

(II) A (source or non-source) cell that *receives* a basic message becomes black (if it is white).

(II') When a cell sends a message, it increments its s-counter by one, $\sigma_i.c = \sigma_i.c+1$; when a cell receives a message, it decrements its s-counter by one, $\sigma_i.c = \sigma_i.c-1$.

(III) When the source cell becomes *passive* in the basic algorithm, it sends a white s-token with $s\text{-}sum = 0$ to start the first round (this is done only once).

Control layer of Safra's algorithm:

(IV) A non-source cell, σ_i, only forwards the s-token with $s\text{-}sum = s\text{-}sum + \sigma_i.c$ when it is *passive* in the basic algorithm.

(V) When a black non-source cell forwards the s-token, the s-token becomes black (if it is white).

(VI) Each (source or non-source) cell becomes white (if it is black) immediately after forwarding the s-token.

(VII) When the s-token returns to the source cell, the source cell waits until it is *passive* in the basic algorithm:

- (a) if the s-token and the source cell are white and $s\text{-}sum = 0$, the source cell knows the termination;
- (b) otherwise, the source cell sends a white s-token with $s\text{-}sum = \sigma_s.c$, to start another round.

Rules (IV) and (VII) of Safra's algorithm can be only applied when a cell would become *passive* (§ 1.2) in the basic algorithm. In xP systems, this is achieved by the parallel composition with interaction (§ 2.3.7).

Example 3.14. Figure 3.17 shows how to apply Safra's algorithm to AsynchBFS in one *asynchronous* scenario in Section 3.4.3. The graph, G, contains a ring, $\sigma_1.\sigma_2.\sigma_3.\sigma_4.\sigma_1$. Initially, each cell, σ_i, sets its s-counter, $\sigma_i.c = 0$.

(a) The source cell, σ_1, broadcasts its v-token, $\sigma_1.c = 0 + 3 = 3$.

(b) The source cell, σ_1, sends a white s-token with $s\text{-}sum = \sigma_1.c = 3$ to start round 1.

Cell σ_2 receives the v-token from σ_1 and becomes black, $\sigma_2.c = 0 - 1 = -1$; then σ_2 sends its v-tokens to σ_3 and σ_4, $\sigma_2.c = -1 + 2 = 1$.

(c) On receiving the white s-token from σ_1, black σ_2 sends a black s-token with $s\text{-}sum = 3 + 1 = 4$ to σ_3 and becomes white.

Cell σ_3 receives the v-token from σ_2 and becomes black, $\sigma_3.c = 0 - 1 = -1$; then σ_3 sends its v-tokens to σ_1 and σ_4, $\sigma_3.c = -1 + 2 = 1$.

(d) Cells σ_1 receives the v-token from σ_3 and becomes black, $\sigma_1.c = 3 - 1 = 2$.

On receiving the black s-token from σ_2, black σ_3 forwards the black s-token with $s\text{-}sum = 4 + 1 = 5$ to σ_4 and becomes white.

Cell σ_4 receives the v-token from σ_3 and becomes black, $\sigma_4.c = 0 - 1 = -1$; then σ_4 sends its v-token to σ_1 and σ_2, $\sigma_4.c = -1 + 2 = 1$.

(e) Cells σ_1 and σ_2 receive the v-tokens from σ_4 and σ_2 becomes black (σ_1 remains black), $\sigma_1.c = 2 - 1 = 1$ and $\sigma_2.c = 1 - 1 = 0$.

On receiving the black s-token from σ_3, black σ_4 forwards the black s-token with $s\text{-}sum = 5 + 1 = 6$ to σ_1 and becomes white.

Assume that the v-tokens from σ_1 to σ_3, from σ_1 to σ_4 and from σ_2 to σ_4 arrive.

(f) The black source cell, σ_1 receives the black s-token with $s\text{-}sum = 6$, so it sends again a white s-token with $s\text{-}sum = \sigma_1.c = 1$ to σ_2 to start round 2 and becomes white.

Cell σ_3 receives the v-token from σ_1 and becomes black, $\sigma_3.c = 1 - 1 = 0$; then σ_3 updates its distance and sends distance notifications to σ_2 and σ_4, $\sigma_3.c = 0 + 2 = 2$.

Cell σ_4 receives the v-tokens from σ_1 and σ_2 and becomes black, $\sigma_4.c = 1 - 2 = -1$; then σ_4 updates its distance and sends distance notifications to σ_2 and σ_3, $\sigma_4.c = -1 + 2 = 1$.

(g) Cell σ_2 receives σ_3 and σ_4's distance notifications and remains black, $\sigma_2.c = 0 - 2 = -2$; on receiving the white s-token from σ_1, black σ_2 sends a black s-token with $s\text{-}sum = 1 + (-2) = -1$ to σ_3 and becomes white.

Black cells σ_3 and σ_4 receive each other's distance notifications, $\sigma_3.c = 2 - 1 = 1$ and $\sigma_4.c = 1 - 1 = 0$.

(h) On receiving the black s-token from σ_2, black σ_3 sends a black s-token with $s\text{-}sum = -1 + 1 = 0$ to σ_4 and becomes white.

(i) On receiving the black s-token from σ_3, black σ_4 sends a black s-token with $s\text{-}sum = 0 + 0 = 0$ to σ_1 and becomes white.

(j) The white source cell, σ_1 receives the black s-token with $s\text{-}sum = 0$ from σ_4, so it sends again a white s-token with $s\text{-}sum = \sigma_1.c = 1$ to σ_2 to start round 3.

(k, l, m) The white token travels around the ring.

(n) Finally, the white source cell, σ_1, receives a white s-token with $s\text{-}sum = 0$ and thus knows the termination.

The correctness and liveness theorem in Tel's book [62] indicates that the detection latency of Safra's algorithm is *one or two rounds*. Consider round r when the basic algorithm terminates.

(1) If the s-token makes all cells white in round r, then round $r + 1$ detects termination.

(2) Otherwise (for the cells that become black after the s-token passes by, the s-token can not make them white in the same round r), the s-token makes all cells white in round $r + 1$ and round $r + 2$ detects termination.

Thus, the runtime complexity of Safra's algorithm is $O(n)$. Example 3.14 shows case (1), in which the basic algorithm terminates in round 2 and round 3 detects termination.

xP Specification 3.13: Control layer of Safra's algorithm

Input: Assumptions of the basic algorithm are made: (I) initially, all cells contain no b', x or x'; (II) for each received message, a cell generates one b' if it does not contain b' and generates one x; (II') for each sent messsage, a cell generates one x'; (III) the source cell, σ_s, sends a loopback message of one t when it starts computation. Additionally, each cell, σ_i, contains its ring successor pointer, r_k.

Output: The source cell, σ_s, contains one g, indicating it knows the algorithm termination.

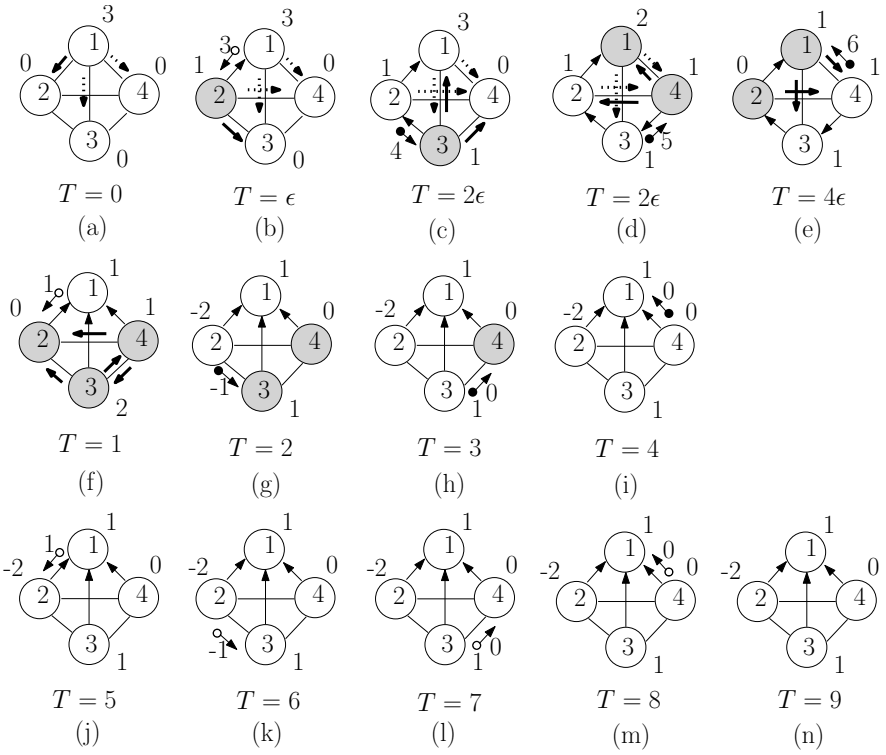


Figure 3.17: An example of AsynchBFS+Safra in one possible asynchronous run. Dotted thick arrows near edges: messages still in transit; edges with arrows: virtual spanning tree child-parent arcs; thick arrows near edges indicate basic messages (a distance notification, or a v-token = a visit token + a distance notification); thin arrows with white or black circles near edges: white or black s-tokens respectively; the number beside each cell: its s-counter; the number beside the s-token: s-sum.

Symbols and states

Our xP Specification 3.13 of Safra's algorithm uses the same state and symbols as xP Specification 3.11 of the DFG algorithm, except that w and b respectively indicate white and black s-tokens instead of d-tokens and t indicates that a cell must next send an s-token instead of a d-token. Also, cell σ_i uses the following additional symbols:

- x' and x indicate cell its signed integer s-counter: $\sigma_i.c = x' - x$;
- z and z' indicate the s-sum: $s\text{-}sum = z' - z$.

For example, row 3, Fig.(c), of Table 3.23 shows that σ_3 contains symbols xx'^2, so σ_3's s-counter is the multiplicity of x' minus the multiplicity of x, i.e. $2 - 1 = 1$, row 4, Fig.(d), shows that σ_3 receives a message containing z'^4, so the s-sum is the multiplicity of z' minus the multiplicity of z, i.e. $4 - 0 = 4$.

The matrix R of xP Specification 3.13 consists of only one vector.

1. Detection

1.1. $S_1\ z\ z' \ \rightarrow_{\texttt{max}}\ S_1 \mid s$

1.2. $S_1\ w \ \rightarrow_{\texttt{min}}\ S_1\ g \mid s \neg b'\ z\ z'$

1.3. $S_1\ w \ \rightarrow_{\texttt{min}}\ S_1\ t \mid \neg\ g$

1.4. $S_1\ b \ \rightarrow_{\texttt{min}}\ S_1\ t \mid \neg\ g$

1.5. $S_1\ z \ \rightarrow_{\texttt{max}}\ S_1 \mid s$

1.6. $S_1\ z' \ \rightarrow_{\texttt{max}}\ S_1 \mid s$

1.7. $S_1\ x\ x' \ \rightarrow_{\texttt{max}}\ S_1 \mid t$

1.8. $S_1\ z\ z' \ \rightarrow_{\texttt{max}}\ S_1 \mid t$

1.9. $S_1\ x \ \rightarrow_{\texttt{min.max}}\ S_1\ (x)\ (z) \updownarrow_j \mid t\ r_j$

1.10. $S_1\ x' \ \rightarrow_{\texttt{min.max}}\ S_1\ (x')\ (z') \updownarrow_j \mid t\ r_j$

1.11. $S_1\ z \ \rightarrow_{\texttt{min.max}}\ S_1\ (z) \updownarrow_j \mid t\ r_j$

1.12. $S_1\ z' \ \rightarrow_{\texttt{min.max}}\ S_1\ (z') \updownarrow_j \mid t\ r_j$

1.13. $S_1\ t\ b' \ \rightarrow_{\texttt{min.min}}\ S_1\ (w) \updownarrow_j \mid s\ r_j$

1.14. $S_1\ t \ \rightarrow_{\texttt{min.min}}\ S_1\ (w) \updownarrow_j \mid r_j \neg b'$

1.15. $S_1\ t\ b' \ \rightarrow_{\texttt{min.min}}\ S_1\ (b) \updownarrow_j \mid r_j$

Rule explanations

The xP rules correspond to the detection rules of the control layer of Safra's algorithm.

(III) Rules 1.3–4, 1.7–12: when cell σ_i receives a white or black s-token, w or b, it generates one t, indicating it must next send an s-token (rules 1.3–4). Then σ_i computes $s\text{-}sum = s\text{-}sum + \sigma_i.c$ by sending x's with z's and x''s with z''s (rules 1.7–12) to its ring successor, r_j, with the s-token in the same step (as follows).

If σ_i is white, $\neg\ b'$, it sends a white s-token, w, to its ring successor, r_j, and deletes t (rule 1.14).

(IV) Rule 1.15: otherwise (if σ_i is black, b'), it sends a black s-token, b, to its ring successor, r_j, becomes white by erasing b' and deletes t.

(V) Rule 1.15: as discussed in (IV).

(VI) Rules 1.1–7, 1.9–10, 1.13–14: consider the source cell, σ_s, which receives the s-token, w or b.

(a) Rules 1.1–2: if σ_s is white, $\neg\ b'$, and receives w with $s\text{-}sum = 0$, $\neg\ z\ z'$, it generates one g, indicating that it knows the termination.

(b) Rules 1.3–7, 1.9–10, 1.13–14: otherwise, $\neg\, g$, σ_s generates one t (rules 1.3–4) and then sends w (rules 1.13–14) to start another round with $s\text{-}sum = \sigma_s.c$ (rules 1.5–7, 1.9–10) and deletes t. Also, if σ_s is black, b', it becomes white by erasing b' (rule 1.13).

Algorithm AsynchBFS+Safra

Example 3.14 illustrates the evolution of AsynchBFS+Safra in one *asynchronous* mode.

xP Specification 3.14: AsynchBFS+Safra

Input: Same as in xP Specification 3.8. Additionally, each cell, σ_i, contains its ring successor pointer, r_k.

Output: Same as in xP Specification 3.8. All ring successor pointer symbols are intact. Additionally, the source cell, σ_s, contains one g, indicating it knows the algorithm termination.

Symbols and states

The modified AsynchBFS, Π_1^*, uses the same symbols as xP Specification 3.8 of AsynchBFS, plus a few more specific to Safra's algorithm. Cell σ_i uses the following additional symbols:

- r_j indicates a ring successor, σ_j;
- b' indicates that it is black (white if it does not contain b');
- x' and x indicate its signed integer s-counter;
- t indicates that it must next send an s-token.

The matrix R of xP Specification 3.14 consists of only one vector.

1. Main search and detection

$\overline{\Pi_1^*}$: rules of modified AsynchBFS by replacing S_2 with $\Theta(S_2, Y)$, where boxed rules correspond to additions to AsynchBFS for Safra's algorithm.

1.1. $\boxed{\Theta(S_2,Y) \;\rightarrow_{\texttt{min.min}}\; \Theta(S_2,Y)\ (t)\ f_i\ u_i(\lambda) \mid \iota_i\ s \neg\ v}$

1.2. $\Theta(S_2,Y)\ f_j \;\rightarrow_{\texttt{min.min}}\; \Theta(S_2,Y)\ f\ v\ p_j \mid \iota_i \neg\ v$

1.3. $\Theta(S_2,Y) \;\rightarrow_{\texttt{max.min}}\; \Theta(S_2,Y)\ (f_i) \updownarrow_j \mid \iota_i\ f\ n_j\ d_i(X) \neg\ p_j$

1.4. $\Theta(S_2,Y)\ f \;\rightarrow_{\texttt{min}}\; \Theta(S_2,Y)$

1.5. $\Theta(S_2,Y)\ f_j \;\rightarrow_{\texttt{max.max}}\; \Theta(S_2,Y) \mid v$

1.6. $\boxed{\Theta(S_2,Y) \;\rightarrow_{\texttt{min.min}}\; \Theta(S_2,Y)\ b' \mid u_j(X) \neg\ b'}$

1.7. $\boxed{\Theta(S_2, Y)\ u_j(\lambda)\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2, Y)\ m_j(Xc)}$

1.8. $\boxed{\Theta(S_2, Y)\ u_j(X)\ \rightarrow_{\texttt{max.min}}\ \Theta(S_2, Y)\ x\ m_j(Xc)}$

1.9. $\Theta(S_2, Y)\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2, Y)\ m'_j(X) \mid m_j(X)$

1.10. $\Theta(S_2, Y)\ m'_j(XY)\ \rightarrow_{\texttt{max.min}}\ \Theta(S_2, Y)\ m'_j(X) \mid m_j(X)$

1.11. $\Theta(S_2, Y)\ m'_j(XY)\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2, Y) \mid \iota_i\ d_i(X)$

1.12. $\Theta(S_2, Y)\ d_i(XY)\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2, Y) \mid \iota_i\ m'_j(X)$

1.13. $\Theta(S_2, Y)\ m'_j(X)\ p_k\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2, Y)\ p_j\ (d_i(X))\ d'_i(X)\ m \mid \iota_i$

1.14. $\boxed{\Theta(S_2, Y)\ \rightarrow_{\texttt{max.min}}\ \Theta(S_2, Y)\ x'\ (u_i(X)) \updownarrow_j\ \mid \iota_i\ m\ n_j\ d'_i(X) \neg p_j}$

1.15. $\Theta(S_2, Y)\ m_j(X)\ \rightarrow_{\texttt{max.max}}\ \Theta(S_2, Y)$

1.16. $S_2\ d'_i(X)\ \rightarrow_{\texttt{min.min}}\ S_2 \mid \iota_i$

1.17. $\Theta(S_2, Y)\ m\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2, Y)$

$\overline{\Pi_2}$: rules of the control layer of Safra's algorithm by replacing S_1 with $\Theta(X, S'_1)$.

Initial and final configurations

Table 3.22 shows the initial and final configurations of xP Specification 3.14 for Figure 3.17.

Table 3.22: Initial and final configurations of xP Specification 3.14 for Figure 3.17.

Time	σ_1	σ_2	σ_3	σ_4
0	$\Theta(S_2, S'_1)\ s\ \iota_1\ r_2$ $n_2\ n_3\ n_4$	$\Theta(S_2, S'_1)\ \iota_2\ r_3$ $n_1\ n_3\ n_4$	$\Theta(S_2, S'_1)\ \iota_3\ r_4$ $n_1\ n_2\ n_4$	$\Theta(S_2, S'_1)\ \iota_4\ r_1$ $n_1\ n_2\ n_3$
9	$\Theta(S_2, S'_1)\ s\ \iota_1\ r_2\ v\ p_1$ $x'\ n_2\ n_3\ n_4\ \boldsymbol{g}$	$\Theta(S_2, S'_1)\ \iota_2\ r_3\ v\ p_1$ $x^2\ n_1\ n_3\ n_4$	$\Theta(S_2, S'_1)\ \iota_3\ r_4\ v\ p_2$ $x'\ n_1\ n_2\ n_4$	$\Theta(S_2, S'_1)\ \iota_4\ r_1\ v\ p_3$ $n_1\ n_2\ n_3$

Rule explanations

An AsynchBFS message is either (1) a v-token (visit token + distance notification), $f_j\ u_j(X)$ or (2) a distance notification, $u_j(X)$, so every message contains one symbol $u_j(X)$. The xP rules of the modified AsynchBFS correspond to the detection rules of additions to the basic algorithm for Safra's algorithm.

(I) Initially, all cells contain no b', x or x'.

(II) Rule 1.6: cell σ_i generates one b' if no b' exists for each received $u_j(X)$.

(II') Rules 1.7–8 and 1.14: cell σ_i generates one x for each received $u_j(X)$, except for σ_s's self-generated $u_s(\lambda)$ at the start of the algorithm (rule 1.7), and σ_i generates one x' for each sent $u_j(X)$.

(III) Rule 1.1: the source cell, σ_s, sends a loopback message of one t, indicating that it must next send an s-token.

Partial traces

Table 3.23 shows partial traces of xP Specification 3.14 for cell σ_3 in Figure 3.17, highlighting the symbols used for Safra's algorithm. Omitted symbols (...) are $\iota_3\ n_1\ n_2\ n_4$ (see Table 3.22).

Table 3.23: Partial traces of xP Specification 3.14 for cell σ_3 in Figure 3.17.

Fig.	Evolution	Content
(a)		...
(b)		...
(c)	$\{f_2\ u_2(c^2)\} \Rightarrow v\ p_2\ d_3(c^3)\ \boldsymbol{b'\ x\ x'^2}\ \{f_3\ u_3(c^3)\}_{1,4}$	$v\ p_2\ d_3(c^3)\ \boldsymbol{b'\ x\ x'^2} \ldots$
(d)	$\{\boldsymbol{b\ z'^4}\}\ \boldsymbol{b'\ x\ x'^2} \Rightarrow \{\boldsymbol{x'}\}\ \{\boldsymbol{b\ z'^5}\}_4$	$v\ p_2\ d_3(c^3) \ldots$
(e)		$v\ p_2\ d_3(c^3)\ \boldsymbol{x'} \ldots$
(f)	$\{f_1\ u_1(c)\}\ p_2\ d_3(c^3) \Rightarrow p_1\ d_3(c^2)\ \boldsymbol{b'\ x\ x'^2}\ \{u_3(c^2)\}_{2,4}$	$v\ p_1\ d_3(c^2)\ \boldsymbol{b'\ x\ x'^3} \ldots$
(g)	$\{u_4(c^2)\} \Rightarrow\ \boldsymbol{x}$	$v\ p_1\ d_3(c^2)\ \boldsymbol{b'\ x^2\ x'^3} \ldots$
(h)	$\{\boldsymbol{b\ z^2\ z'}\}\ \boldsymbol{b'\ x^2\ x'^3} \Rightarrow \{\boldsymbol{x'}\}\ \{\boldsymbol{b\ z\ z'}\}_4$	$v\ p_1\ d_3(c^2) \ldots$
(i) (j) (k)		$v\ p_1\ d_3(c^2)\ \boldsymbol{x'} \ldots$
(l)	$\{\boldsymbol{w\ z^2\ z'}\}\ \boldsymbol{x'} \Rightarrow \{\boldsymbol{x'}\}\ \{\boldsymbol{w\ z\ z'}\}_4$	$v\ p_1\ d_3(c^2) \ldots$
(m)		$v\ p_1\ d_3(c^2)\ \boldsymbol{x'} \ldots$

3.6.3 The Dijkstra-Scholten Algorithm

The Dijkstra-Scholten algorithm detects termination by *dynamically* maintaining a *computation tree*, i.e. a dynamic partial spanning tree, which requires an underlying network with *duplex channels*.

This algorithm looks for a situation where, for *each* cell, (1) it is *passive* and (2) all its outgoing basic messages are acknowledged (i.e. its outgoing channels are empty).

Assume that the basic algorithm is extended with ingredients for the Dijkstra-Scholten algorithm: *ds-parents*, *ds-counters* and *ds-acknowledgments*. The Dijkstra-Scholten algorithm detects termination of the basic algorithm based on the following detection rules.

Additions to the basic algorithm:

(I) Initially, each cell initialises its ds-counter to 0 and the source cell, σ_s, sets its ds-parent as itself.

(II) When a (source or non-source) cell, σ_i, sends a basic message to a cell, it increments its ds-counter by one, $\sigma_i.c = \sigma_i.c + 1$.

(III) When a (source or non-source) cell, σ_i, receives a basic message from cell σ_j,

(a) if σ_i has no ds-parent, it records its ds-parent as σ_j (it will later send a ds-acknowledgment to σ_j);

(b) if σ_i has its ds-parent, it sends a ds-acknowledgment to σ_j.

Control layer of the Dijkstra-Scholten algorithm:

(IV) When a (source or non-source) cell, σ_i, receives a ds-acknowledgment, it decrements its ds-counter by one, $\sigma_i.c = \sigma_i.c - 1$, and deletes the ds-acknowledgment.

(V) When a (source or non-source) cell satisfies two local conditions that (1) it is passive and (2) its ds-counter reaches zero:

(a) if it is a non-source cell, σ_i, it sends a ds-acknowledgment to its ds-parent and deletes its ds-parent;

(b) if it is the source cell, σ_s, it knows the algorithm termination.

Rule (V) of the Dijkstra-Scholten algorithm can be only applied when a cell would become *passive* (§ 1.2) in the basic algorithm. In xP systems, this is achieved by the parallel composition with interaction (§ 2.3.7).

According to rule (II), a cell increments its ds-counter by one for each message it sent to a cell. If a cell broadcasts a message to all its neighbours (such as in AsynchND, § 3.5), it increments its ds-counter by the number of its neighbours. This rule implies that a cell knows all its neighbours or the number of them.

After a cell deletes its ds-parent, it can set its ds-parent again when it later receives a basic message. Thus, the computation tree can grow and shrink repeatedly in different ways [42] as a partial spanning tree of the underlying network digraph.

For each basic message from σ_j to σ_k, the Dijkstra-Scholten algorithm sends exactly one control message from σ_k to σ_j. Thus, the message complexity is M, where M is the number of messages of the basic algorithm.

When this basic algorithm terminates, a computation tree is built and this computation tree takes at most $n - 1$ time units to shrink to the source cell. Thus, the time complexity of the Dijkstra-Scholten algorithm is $O(n)$ [63].

xP Specification 3.15: Control layer of the Dijkstra-Scholten algorithm

Input: Assumptions of the basic algorithm are made: (I) the source cell, σ_s, generates one p'_s when it starts computation; (II) for each message sent to a neighbour cell, a cell generates one w; (III) a cell that receives a basic message from σ_j records its ds-parent as p'_j if it does not contain p'_k or sends an a_i to σ_j if it contains p'_k.

Output: The source cell, σ_s, contains one g, indicating it knows the algorithm termination.

Symbols and states

Cell σ_i uses the following symbols:

- p'_j indicates its ds-parent, σ_j;
- a_i is its ds-acknowledgment;
- w is used for its ds-counter;
- g indicates that it knows the algorithm termination.

The control layer of the Dijkstra-Scholten algorithm uses only one state, S_1.

The matrix R of xP Specification 3.15 consists of only one vector.

1. Detection

1.1. $S_1\ a_j\ w\ \rightarrow_{\texttt{max.min}}\ S_1$

1.2. $S_1\ p'_j\ \rightarrow_{\texttt{min.min}}\ S_1\ g \mid s \neg w$

1.3. $S_1\ p'_j\ \rightarrow_{\texttt{min.min}}\ S_1\ (a_i) \updownarrow_j \mid \iota_i\ w$

Rule explanations

The xP rules correspond to the detection rules of the control layer of the Dijkstra-Scholten algorithm.

(IV) Rule 1.1: when cell σ_i receives a ds-acknowledgment, a_j, it decrements its ds-counter by deleting one w and removes a_j.

(V) Rules 1.2–3: consider cell σ_i, which receives all ds-acknowledgments, $\neg\ w$.

(a) Rule 1.3: if σ_i is a non-source cell, it sends a ds-acknowledgment, a_i, to its ds-parent, p'_j, and deletes p'_j.

(b) Rule 1.2: if σ_i is the source cell, σ_s, it generates one g, indicating that it knows the termination.

Algorithm AsynchBFS+ Dijkstra-Scholten (AsynchBFS+DS)

To solve the termination detection problem of AsynchBFS, we apply the Dijkstra-Scholten algorithm to AsynchBFS so that the source cell knows the algorithm termination. Example 3.15 shows one possible asynchronous evolution of AsynchBFS+DS.

Example 3.15. Figure 3.18 shows the evolution of AsynchBFS+DS in the *asynchronous* scenario in Section 3.4.3.

(a) At the start, the source cell, σ_1, broadcasts its v-token and sets its ds-parent as itself and $\sigma_1.c = 3$.

(b) Cell σ_2 receives the v-token from σ_1, so it records its ds-parent as σ_1 and sends its v-token to σ_3 and σ_4, $\sigma_2.c = 0 + 2 = 2$.

(c) Cell σ_3 receives the v-token from σ_2, so it records its ds-parent as σ_2 and sends its v-token to σ_1 and σ_4, $\sigma_3.c = 0 + 2 = 2$.

(d) Cell σ_1 receives the v-token from σ_3 and sends a ds-acknowledgment to σ_3. Cell σ_4 receives the v-token from σ_3, so it records its ds-parent as σ_3 and sends its v-token to σ_1 and σ_2, $\sigma_4.c = 0 + 2 = 2$.

(e) Cells σ_1 and σ_2 receive the v-tokens from σ_4 and send ds-acknowledgments to σ_4. Assume that v-token and ds-acknowledgment from σ_1 to σ_3, the v-token from σ_1 to σ_4 and the v-token from σ_2 to σ_4 arrive.

(f) Cell σ_3 receives the v-token from σ_1 and sends a ds-acknowledgment to σ_1; σ_3 updates its distance and sends distance notifications to σ_2 and σ_4, $\sigma_3.c = 2+2 = 4$; σ_3 receives σ_1's ds-acknowledgment, $\sigma_3.c = 4 - 1 = 3$.

Cell σ_4 receives the v-tokens from σ_1 and σ_2 and sends back ds-acknowledgments to σ_1 and σ_2; σ_4 updates its distance and sends distance notifications to σ_2 and σ_3, $\sigma_4.c = 2+2 = 4$; σ_4 receives σ_1 and σ_2's ds-acknowledgments, $\sigma_4.c = 4-2 = 2$.

(g) Cell σ_1 receives σ_3 and σ_4's acknowledgments, $\sigma_1.c = 3 - 2 = 1$. Cell σ_2 receives σ_3 and σ_4's distance notifications and sends back ds-acknowledgments; σ_2 receives σ_4's ds-acknowledgment, $\sigma_2.c = 2 - 1 = 1$. Cells σ_3 and σ_4 receive each other's distance notifications and send back ds-acknowledgments.

(h) Cell σ_3 receives σ_2 and σ_4's ds-acknowledgments, $\sigma_3.c = 3 - 2 = 1$. Cell σ_4 receives σ_2 and σ_3's ds-acknowledgments, $\sigma_4.c = 2 - 2 = 0$, and thus σ_4 sends a ds-acknowledgment to its ds-parent, σ_3.

(i) Cell σ_3 receives σ_4's ds-acknowledgment, $\sigma_3.c = 1 - 1 = 0$, and thus σ_3 sends a ds-acknowledgment to its ds-parent, σ_2.

(j) Cell σ_2 receives σ_3's ds-acknowledgment, $\sigma_2.c = 1 - 1 = 0$, and thus σ_2 sends a ds-acknowledgment to its ds-parent, σ_1.

(k) Finally, the source cell, σ_1, receives all ds-acknowledgments and thus knows the algorithm termination.

Although not shown in Example 3.15, our implementation of the Dijkstra-Scholten algorithm also has the race condition problem (§ 3.2): a cell that has no ds-parent may receive several basic messages simultaneously, but only of the sending cells can become the cell's ds-parent. This can be solved by using the `min.min` mode, in the same way as the Echo algorithm.

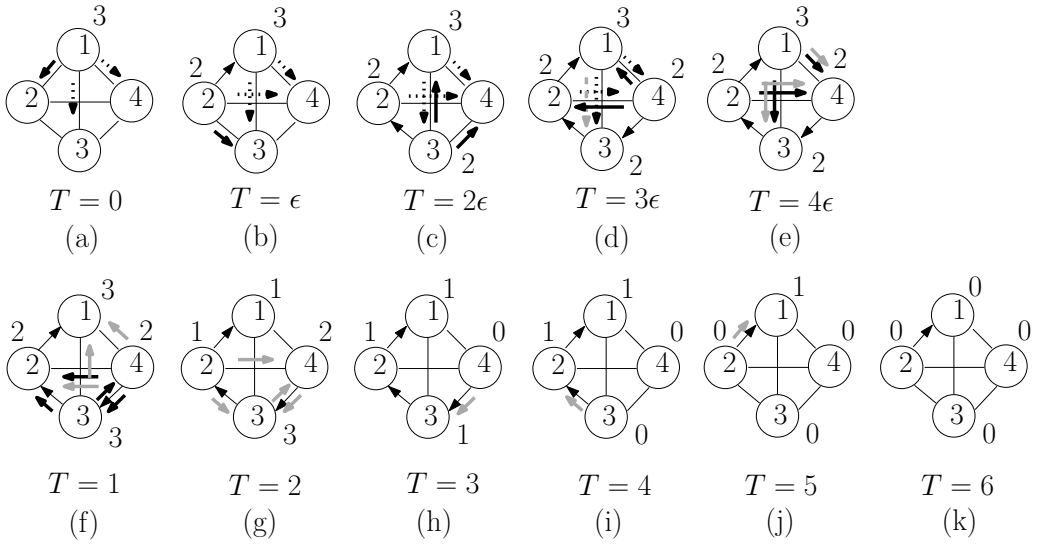


Figure 3.18: An example of AsynchBFS+DS in one possible asynchronous run. Dotted thick arrows near edges: messages still in transit; edges with arrows: computation tree child-parent arcs; thick arrows near edges: basic messages (a distance notification, or a v-token = a visit token + a distance notification); gray arrows near edges: ds-acknowledgments; the number beside each cell: its ds-counter.

xP Specification 3.16: AsynchBFS+DS

Input: Same as in xP Specification 3.8.

Output: Same as in xP Specification 3.8. Additionally, the source cell, σ_s, contains one g, indicating it knows the algorithm termination.

Symbols and states

The modified AsynchBFS, Π_1^*, uses the same symbols as xP Specification 3.8 of AsynchBFS, plus a few more specific. Cell σ_i uses the following additional symbols:

- p'_j indicates its ds-parent, σ_j;
- a_i is its ds-acknowledgment;
- w is used for its ds-counter.

The matrix R of xP Specification 3.16 consists of only one vector.

1. Main search and detection

$\overline{\Pi_1^*}$: rules of modified AsynchBFS by replacing S_2 with $\Theta(S_2, Y)$, where boxed rules correspond to additions to AsynchBFS for the Dijkstra-Scholten algorithm.

1.1. $\boxed{\Theta(S_2,Y) \ \rightarrow_{\texttt{min.min}} \ \Theta(S_2,Y) \ f_i \ u_i(\lambda) \mid \iota_i \ s \ \neg \ v}$

1.2. $\Theta(S_2,Y)\ f_j\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2,Y)\ f\ v\ p_j \mid \iota_i \neg v$

1.3. $\Theta(S_2,Y)\ \rightarrow_{\texttt{max.min}}\ \Theta(S_2,Y)\ (f_i)\updownarrow_j \mid \iota_i\ f\ n_j \neg p_j$

1.4. $\Theta(S_2,Y)\ f\ \rightarrow_{\texttt{min}}\ \Theta(S_2,Y)$

1.5. $\Theta(S_2,Y)\ f_j\ \rightarrow_{\texttt{max.max}}\ \Theta(S_2,Y) \mid v$

1.6. $\boxed{\Theta(S_2,Y)\ u_j(X)\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2,Y)\ p'_j\ m_j(Xc) \neg p'_k}$

1.7. $\boxed{\Theta(S_2,Y)\ u_j(X)\ \rightarrow_{\texttt{max.min}}\ \Theta(S_2,Y)\ m_j(Xc)\ (a_i)\updownarrow_j \mid \iota_i\ p'_k}$

1.8. $\Theta(S_2,Y)\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2,Y)\ m'_j(X) \mid m_j(X)$

1.9. $\Theta(S_2,Y)\ m'_j(XY)\ \rightarrow_{\texttt{max.min}}\ \Theta(S_2,Y)\ m'_j(X) \mid m_j(X)$

1.10. $\Theta(S_2,Y)\ m'_j(XY)\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2,Y) \mid \iota_i\ d_i(X)$

1.11. $\Theta(S_2,Y)\ d_i(XY)\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2,Y) \mid \iota_i\ m'_j(X)$

1.12. $\Theta(S_2,Y)\ m'_j(X)\ p_k\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2,Y)\ d_i(X)\ p_j\ m \mid \iota_i$

1.13. $\boxed{\Theta(S_2,Y)\ \rightarrow_{\texttt{max.min}}\ \Theta(S_2,Y)\ w\ (u_i(X))\updownarrow_j \mid \iota_i\ m\ n_j\ d_i(X) \neg p_j}$

1.14. $\Theta(S_2,Y)\ m_j(X)\ \rightarrow_{\texttt{max.max}}\ \Theta(S_2,Y)$

1.15. $\Theta(S_2,Y)\ d'_i(X)\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2,Y) \mid \iota_i$

1.16. $\Theta(S_2,Y)\ m\ \rightarrow_{\texttt{min.min}}\ \Theta(S_2,Y)$

$\overline{\Pi_2}$: rules of the control layer of the Dijkstra-Scholten algorithm by replacing S_1 with $\Theta(X,S'_1)$.

Initial and final configurations

Table 3.24 shows the initial and final configurations of xP Specification 3.16 for Figure 3.18.

Table 3.24: Initial and final configurations of xP Specification 3.16 for Figure 3.18.

Time	σ_1	σ_2	σ_3	σ_4
0	$\Theta(S_2,S'_1)\ s\ \iota_1\ n_2\ n_3\ n_4$	$\Theta(S_2,S'_1)\ \iota_2\ n_1\ n_3\ n_4$	$\Theta(S_2,S'_1)\ \iota_3\ n_1\ n_2\ n_4$	$\Theta(S_2,S'_1)\ \iota_4\ n_1\ n_2\ n_3$
6	$\Theta(S_2,S'_1)\ s\ \iota_1\ v\ p_1$ $d_1(c)\ n_2\ n_3\ n_4$ ***g***	$\Theta(S_2,S'_1)\ \iota_2\ v\ p_1$ $d_2(c^2)\ n_1\ n_3\ n_4$	$\Theta(S_2,S'_1)\ \iota_3\ v\ p_1$ $d_3(c^2)\ n_1\ n_2\ n_4$	$\Theta(S_2,S'_1)\ \iota_4\ v\ p_1$ $d_4(c^2)\ n_1\ n_2\ n_3$

Rule explanations

As discussed before, an AsynchBFS message is either (1) a v-token (visit token + distance notification), f_j and $u_j(X)$ or (2) a distance notification, $u_j(X)$, so every message contains one symbol $u_j(X)$. The rules of the modified AsynchBFS algorithm correspond to the detection rules of additions to the basic algorithm for the Dijkstra-Scholten algorithm.

(I) Rule 1.1: the source cell, σ_s, generates one v-token, $f_s\ u_s(\lambda)$, which is next used to set its ds-parent, p'_s, in the same step (rule 1.6).

(II) Rule 1.13: cell σ_i increments its ds-counter by generating one w for each $u_i(X)$ sent to a cell.

(III) Rules 1.6–7: consider a cell, σ_i, which receives $u_j(X)$'s.

(a) Rule 1.6: σ_i selects one of the sending cells, by `min.min` mode, as its ds-parent, p'_j, if no p'_k exists.

(b) Rule 1.7: σ_i sends a ds-acknowledgment, a_i, to σ_j if p'_k exists.

Partial traces

Table 3.25 shows partial traces of xP Specification 3.16 for cell σ_3 in Figure 3.18, highlighting the symbols used for the Dijkstra-Scholten algorithm. Omitted symbols $(\dots)$ are $\iota_3\ n_1\ n_2\ n_4$ (see Table 3.24).

Table 3.25: Partial traces of xP Specification 3.16 for cell σ_3 in Figure 3.18.

Fig.	Evolution	Content
(a)		$\dots$
(b)		$\dots$
(c)	$\{f_2\ u_2(c^2)\} \Rightarrow v\ p_2\ d_3(c^3)\ \boldsymbol{p'_2}\ \boldsymbol{w^2}\ \{f_3\ u(c^3)\}_{1,4}$	$v\ p_2\ d_3(c^3)\ \boldsymbol{p'_2}\ \boldsymbol{w^2} \dots$
(d) (e)		$v\ p_2\ d_3(c^3)\ \boldsymbol{p'_2}\ \boldsymbol{w^2} \dots$
(f)	$\{f_1\ u_1(c)\ \boldsymbol{a_1}\}\ p_2\ d_3(c^3)\ \boldsymbol{w} \Rightarrow p_1\ d_3(c^2)\ \boldsymbol{w^2}\ \{u_3(c^2)\}_{2,4}\ \boldsymbol{\{a_3\}_1}$	$v\ p_1\ d_3(c^2)\ \boldsymbol{p'_2}\ \boldsymbol{w^3} \dots$
(g)	$\{u_4(c^2)\} \Rightarrow \boldsymbol{\{a_3\}_4}$	$v\ p_1\ d_3(c^2)\ \boldsymbol{p'_2}\ \boldsymbol{w^3} \dots$
(h)	$\{\boldsymbol{a_2}\ \boldsymbol{a_4}\}\ \boldsymbol{w^2} \Rightarrow$	$v\ p_1\ d_3(c^2)\ \boldsymbol{p'_2}\ \boldsymbol{w} \dots$
(i)	$\{\boldsymbol{a_4}\}\ \boldsymbol{w}\ \boldsymbol{p'_2} \Rightarrow \boldsymbol{\{a_3\}_2}$	$v\ p_1\ d_3(c^2) \dots$
(j)		$v\ p_1\ d_3(c^2) \dots$

Algorithm AsynchND+Dijkstra-Scholten (AsynchND+DS)

AsynchND+DS solves the termination detection problem of AsynchND and will be later used as a preliminary phase for a subsequent algorithm such as Echo, a distributed DFS algorithm or AsynchBFS (§ 3.7).

As mentioned before, the Dijkstra-Scholten algorithm requires that a cell knows the number of its neighbours if a cell uses broadcast in the basic algorithm. AsynchND uses broadcast and thus AsynchND+DS requires that a cell knows its neighbours' count. Moreover, in AsynchND, each cell broadcasts its visit token only *once*, so each cell initialises its ds-counter to its neighbours' count.

Example 3.16. Figure 3.19 shows the evolution of AsynchND+DS in the *asynchronous* scenario in Section 3.5. Initially, each cell, σ_i, sets its ds-counter, $\sigma_i.c = 3$.

(a) The source cell, σ_1, broadcasts its visit token.

(b) Cell σ_2 receives a visit token from σ_1, so it sets its ds-parent as σ_1 and broadcasts it visit token.

(c) The source cell, σ_1, receives the visit token from σ_2, so it sends back a ds-acknowledgment to σ_2. Cell σ_3 receives the visit token from σ_2, so it sets its ds-parent as σ_2 and broadcasts its visit token.

(d) Cells σ_1 and σ_2 receive σ_3's visit tokens, so they send back ds-acknowledgments to σ_3; also, cell σ_2 receives σ_1's ds-acknowledgment, $\sigma_2.c = 3 - 1 = 2$. Cell σ_4 receives the visit token from σ_3, so it sets its ds-parent as σ_3 and broadcasts its visit token.

(e) Cells σ_1 and σ_2 receive σ_4's visit tokens, so they send back ds-acknowledgments to σ_4. Cell σ_3 receives σ_4's visit token, so it sends back a ds-acknowledgment to σ_4; σ_3 receives σ_2's ds-acknowledgment, $\sigma_3.c = 3 - 1 = 2$.

Assume that the visit token and ds-acknowledgment from σ_1 to σ_3, the visit token from σ_1 to σ_4 and the visit token from σ_2 to σ_4 arrive.

(f) Cell σ_3 receives the visit token from σ_1, so it sends back a ds-acknowledgment to σ_1; σ_3 receives σ_1's ds-acknowledgment, $\sigma_3.c = 2 - 1 = 1$. Cell σ_4 receives the visit tokens from σ_1 and σ_2, so it sends back ds-acknowledgments to σ_1 and σ_2; σ_4 receives ds-acknowledgments from σ_1, σ_2 and σ_3, $\sigma_4.c = 3 - 3 = 0$, so it sends a ds-acknowledgment to its ds-parent, σ_3.

(g) Cell σ_1 receives ds-acknowledgments from σ_3 and σ_4, $\sigma_1.c = 3 - 2 = 1$. Cell σ_2 receives σ_4's ds-acknowledgment, $\sigma_2.c = 2 - 1 = 1$. Cell σ_3 receives σ_4's ds-acknowledgment, $\sigma_3.c = 1 - 1 = 0$, so it sends a ds-acknowledgment to its ds-parent, σ_2.

(h) Cell σ_2 receives σ_3's ds-acknowledgment, $\sigma_2.c = 1 - 1 = 0$, so it sends a ds-acknowledgment to its ds-parent, σ_1.

(i) Finally, the source cell, σ_1, receives all ds-acknowledgments and thus knows the algorithm termination.

xP Specification 3.17: AsynchND+Dijkstra-Scholten (AsynchND+DS)

Input: Same as in xP Specification 3.10. Additionally, each cell contains a counter indicator, w^q, where q is its neighbour's count.

Output: Same as in xP Specification 3.10. Additionally, the source cell, σ_s, contains one g, indicating it knows the algorithm termination.

Symbols and states

In order to acknowledge received visit tokens (rather than neighbour pointers, which are represented using the same symbol, n_j, as visit tokens in AsynchND), the modified AsynchND, Π_1^*, uses n'_j for a visit token and n_j for a neighbour pointer. Cell σ_i uses the following additional symbols:

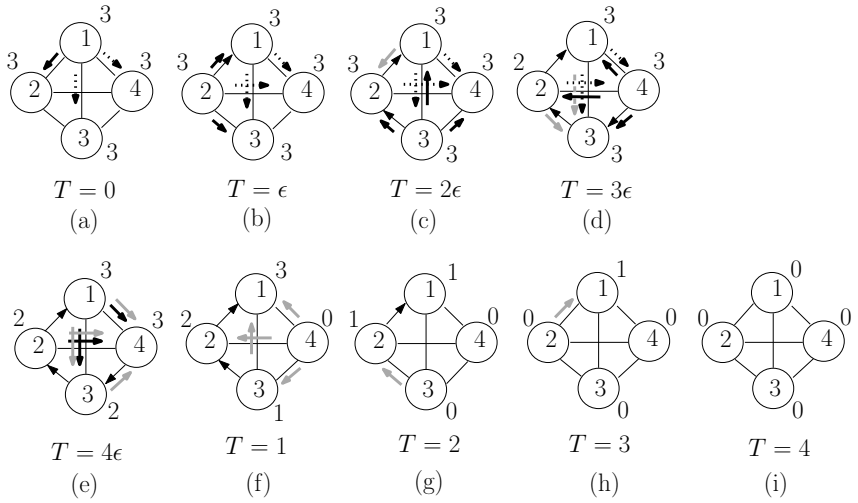


Figure 3.19: An example of AsynchND+DS in one possible asynchronous run. Dotted thick arrows near edges indicate: still in transit; edges with arrows: computation tree child-parent arcs; black arrows near edges: basic messages (visit tokens); gray arrows near edges: ds-acknowledgments; the number beside each cell: its ds-counter.

- n'_i is its visit token;
- n_j records its neighbour, σ_j;
- p'_j indicates its ds-parent, σ_j;
- a_i is its ds-acknowledgment;
- w is used for its ds-counter.

1. An unvisited cell receives visit tokens.

$\overline{\Pi_1^*}$: rules of modified AsynchND by replacing S_i with $\Theta(S_i, Y)$, $i \in \{1, 2\}$, where boxed rules correspond to additions to AsynchND for the Dijkstra-Scholten algorithm.

1.1. $\boxed{\Theta(S_1,Y) \ \rightarrow_{\texttt{min.min}} \ \Theta(S_2,Y)\ p'_i\ (n'_i)\updownarrow \ \mid \iota_i\ s}$

1.2. $\boxed{\Theta(S_1,Y)\ n'_j \ \rightarrow_{\texttt{min.min}} \ \Theta(S_2,Y)\ p'_j\ n_j\ (n'_i)\updownarrow \ \mid \iota_i \neg\ p'_k}$

1.3. $\boxed{\Theta(S_1,Y)\ n'_j \ \rightarrow_{\texttt{max.min}} \ \Theta(S_2,Y)\ n_j\ (a_i)\updownarrow_j \ \mid \iota_i\ p'_k}$

2. A visited cell receives visit tokens.

2.1. $\boxed{\Theta(S_2,Y)\ n'_j \ \rightarrow_{\texttt{max.min}} \ \Theta(S_2,Y)\ n_j\ (a_i)\updownarrow_j \ \mid \iota_i\ p'_k}$

$\overline{\Pi_2}$: rules of the control layer of the Dijkstra-Scholten algorithm by replacing S_1 with $\Theta(X, S_1')$.

Initial and final configurations

Table 3.26 shows the initial and final configurations of xP Specification 3.17 for Figure 3.19 (asynchronous mode).

Table 3.26: Initial and final configurations of xP Specification 3.17 for Figure 3.19.

Time	σ_1	σ_2	σ_3	σ_4
0	$\Theta(S_1, S_1)\ \iota_1\ s\ w^3$	$\Theta(S_1, S_1)\ \iota_2\ w^3$	$\Theta(S_1, S_1)\ \iota_3\ w^3$	$\Theta(S_1, S_1)\ \iota_4\ w^3$
5	$\Theta(S_2, S_1)\ s\ \iota_1\ n_2\ n_3\ n_4\ \boldsymbol{g}$	$\Theta(S_2, S_1)\ \iota_2\ n_1\ n_3\ n_4$	$\Theta(S_2, S_1)\ \iota_3\ n_1\ n_2\ n_4$	$\Theta(S_2, S_1)\ \iota_4\ n_1\ n_2\ n_3$

Rule explanations

The xP rules of the modified AsynchND correspond to the detection rules of additions to the basic algorithm for the Dijkstra-Scholten algorithm.

(I) Rule 1.1: the source cell, σ_s, generates one p_s' when it starts computation.

(II) Initially, each cell contains w^3, which sets its ds-counter to its neighbour's count.

(III) Rules 1.2–3, 2.1: consider a cell, σ_i, which receives n_j''s.

- (a) Rule 1.2: cell σ_i selects one of the sending cells, by `min.min` mode, as its ds-parent, p_j', if no p_k' exists.
- (b) Rules 1.3, 2.1: cell σ_i sends a_i to σ_j if p_k' exists. Specifically, rule 2.1 is needed in this modified version because a visited cell needs to process received visit tokens rather than just receiving them (this is automatically done and no rule is needed, as mentioned in Section 1.2) as in the original basic algorithm, AsynchND.

Partial traces

Table 3.27 shows partial traces of xP Specification 3.17 for cell σ_3 in Figure 3.19 (asynchronous mode), highlighting the symbols used for the Dijkstra-Scholten algorithm. The omitted symbol (...) is ι_3.

Table 3.27: Partial traces of xP Specification 3.17 for cell σ_3 in Figure 3.19.

Fig.	Evolution	Content
(a)		$\boldsymbol{w^3}\ldots$
(b)		$\boldsymbol{w^3}\ldots$
(c)	$\{n_2'\} \Rightarrow n_2\ \boldsymbol{p_2'}\{n_3'\}_{1,2,4}$	$n_2\ \boldsymbol{p_2'}\ \boldsymbol{w^3}\ldots$
(d)		$n_2\ \boldsymbol{p_2'}\ \boldsymbol{w^3}\ldots$
(e)	$\{\boldsymbol{a_2}\ n_4'\}\ \boldsymbol{w} \Rightarrow n_4\ \boldsymbol{\{a_3\}_4}$	$n_2\ n_4\ \boldsymbol{p_2'}\ \boldsymbol{w^2}\ldots$
(f)	$\{\boldsymbol{a_1}\ n_1'\}\ \boldsymbol{w} \Rightarrow n_1\ \boldsymbol{\{a_3\}_1}$	$n_1\ n_2\ n_4\ \boldsymbol{p_2'}\ \boldsymbol{w}\ldots$
(g)	$\{\boldsymbol{a_4}\}\ \boldsymbol{w} \Rightarrow \boldsymbol{\{a_3\}_2}$	$n_1\ n_2\ n_4\ldots$
(h) (i)		$n_1\ n_2\ n_4\ldots$

Algorithm SynchBFS+Dijkstra-Scholten (SynchBFS+DS)

For SynchBFS+DS in the synchronous mode, if each cell's ds-parent is its BFS tree parent, then the ds-acknowledgments are convergecast along the BFS tree branches to the source and the augmenting Dijkstra-Scholten algorithm is actually a supplemented convergecast phase, as mentioned in Section 3.4.2.

Example 3.17. Figure 3.20 shows the evolution of SynchBFS+DS in the *synchronous* mode.

(a) The source cell, σ_1, broadcasts its visit token and sets its ds-counter to 3.

(b) On receiving σ_1's visit token, each of cells σ_2, σ_3 and σ_4 sets its ds-parent as σ_1, sends its visit token to all non-parent neighbours and sets its ds-counter to 2.

(c) On receiving visit tokens, because σ_2, σ_3 and σ_4 have ds-parents, each of them sends back a ds-acknowledgment to the senders of its received visit tokens.

(d) Cells σ_2, σ_3 and σ_4 receive all ds-acknowledgments, i.e. their ds-counters reach 0, so each of them sends a ds-acknowledgment to its ds-parent, σ_1.

(e) The source cell, σ_1, receives all ds-acknowledgments and thus knows the algorithm termination.

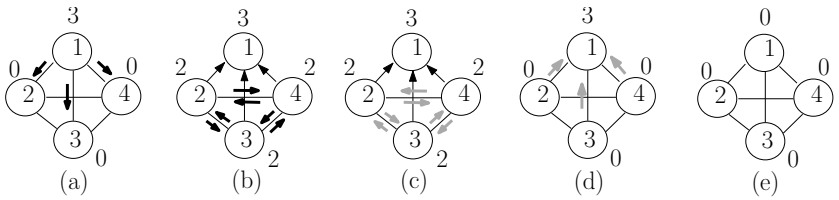


Figure 3.20: An example of SynchBFS+DS in the synchronous mode. Edges with arrows: computation tree arcs; black arrows near edges: basic messages (visit tokens); gray arrows near edges: ds-acknowledgments; the number beside each cell: its ds-counter.

xP Specification 3.18: SynchBFS+Dijkstra-Scholten (SynchBFS+DS)

Input: Same as in xP Specification 3.7.

Output: Same as in xP Specification 3.7. Additionally, the source cell, σ_s, contains one g, indicating it knows the algorithm termination.

Symbols and states

The modified SynchBFS, Π_1^*, uses the same symbols as xP Specification 3.7 of SynchBFS, plus a few more specific to the Dijkstra-Scholten algorithm. Cell σ_i uses the following additional symbols:

- p'_j indicates its ds-parent, σ_j;
- a_i is its ds-acknowledgment;
- w is used for its ds-counter.

The matrix R of xP Specification 3.18 consists of only one vector.

1. Main search and detection

$\overline{\Pi_1^*}$: rules of modified SynchBFS by replacing S_2 with $\Theta(S_2, Y)$, where $\boxed{\text{boxed rules}}$ correspond to additions to SynchBFS for the Dijkstra-Scholten algorithm.

1.1. $\boxed{\Theta(S_2,Y) \ \rightarrow_{\texttt{min.min}} \ \Theta(S_2,Y) \ f_i \mid \iota_i \ s \ \neg \ v}$

1.2. $\boxed{\Theta(S_2,Y) \ f_j \ \rightarrow_{\texttt{min.min}} \ \Theta(S_2,Y) \ f \ v \ p'_j \ p_j \ \neg \ v \ p'_k}$

1.3. $\boxed{\Theta(S_2,Y) \ \rightarrow_{\texttt{max.min}} \ \Theta(S_2,Y) \ w \ (f_i) \updownarrow_k \ \mid \iota_i \ f \ n_k \ \neg \ p_k}$

1.4. $\Theta(S_2,Y) \ f \ \rightarrow_{\texttt{min}} \ \Theta(S_2,Y)$

1.5. $\boxed{\Theta(S_2,Y) \ f_j \ \rightarrow_{\texttt{max.min}} \ \Theta(S_2,Y) \ (a_i) \updownarrow_j \ \mid \iota_i \ v \ p'_k}$

$\overline{\Pi_2}$: rules of the control layer of the Dijkstra-Scholten algorithm by replacing S_1 with $\Theta(X, S'_1)$.

Initial and final configurations

Table 3.28 shows the initial and final configurations of xP Specification 3.18 for Figure 3.20.

Rule explanations

The xP rules of the modified SynchBFS correspond to the detection rules of additions to the basic algorithm for the Dijkstra-Scholten algorithm.

Table 3.28: Initial and final configurations of xP Specification 3.18 for Figure 3.20.

Time	σ_1	σ_2	σ_3	σ_4
0	$\Theta(S_2, S_1')$ s ι_1 n_2 n_3 n_4	$\Theta(S_2, S_1')$ ι_2 n_1 n_3 n_4	$\Theta(S_2, S_1')$ ι_3 n_1 n_2 n_4	$\Theta(S_2, S_1')$ ι_4 n_1 n_2 n_3
4	$\Theta(S_2, S_1')$ s ι_1 v p_1 n_2 n_3 n_4 $\boldsymbol{g}$	$\Theta(S_2, S_1')$ ι_2 v p_1 n_1 n_3 n_4	$\Theta(S_2, S_1')$ ι_3 v p_1 n_1 n_2 n_4	$\Theta(S_2, S_1')$ ι_4 v p_1 n_1 n_2 n_3

(I) Rule 1.1: the source cell, σ_s, generates one f_s, which is next used to set its ds-parent, p_s, in the same step (rule 1.2).

(II) Rule 1.3: cell σ_i generates one w for each f_i sent to a cell.

(III) Rules 1.2 and 1.5: consider a cell, σ_i, which receives f_j's.

(a) Rule 1.2: σ_i selects one of the sending cells, by `min.min` mode, as its ds-parent, p_j', if no p_k' exists.

(b) Rule 1.5: σ_i sends a_i to σ_j if p_k' exists.

Partial traces

Table 3.29 shows partial traces of xP Specification 3.18 for cell σ_3 in Figure 3.20, highlighting the symbols used for the Dijkstra-Scholten algorithm. Omitted symbols (...) are ι_3 n_1 n_2 n_4 (see Table 3.28).

Table 3.29: Partial traces of xP Specification 3.18 for cell σ_3 in Figure 3.20.

Fig.	Evolution	Content
(a)		...
(b)	$\{f_1\} \Rightarrow v$ p_1 $\boldsymbol{p_1'}$ $\boldsymbol{w^2}$ $\{f_3\}_{2,4}$	v p_1 $\boldsymbol{p_1'}$ $\boldsymbol{w^2}$ ...
(c)	$\{f_2$ $f_4\} \Rightarrow \boldsymbol{\{a_3\}_{2,4}}$	v p_1 $\boldsymbol{p_1'}$ $\boldsymbol{w^2}$ ...
(d)	$\{\boldsymbol{a_2}$ $\boldsymbol{a_4}\}$ $\boldsymbol{p_1'}$ $\boldsymbol{w^2} \Rightarrow \boldsymbol{\{a_3\}_1}$	v p_1 ...

3.7 Complete Solutions

This section builds complete partial-explicit terminating xP solutions for all traversal algorithms in this chapter, including a preliminary neighbour discovery phase.

In the synchronous mode, if a subsequent synchronous algorithm starts two time units after SynchND, each cell will have enough time to finish neighbour discovery. Thus, we use SynchND as a preliminary phase for SynchBFS+DFG, The complete xP solution, denoted as SynchND+SynchBFS+DFG, is constructed by sequential composition, $\Pi_1; \Pi_2$ (§ 2.3.7).

1. Π_1: SynchND;
2. Π_2: SynchBFS+DFG.

In the asynchronous mode, AsynchND+DS ensures that the source cell knows when all cells finish neighbour discovery, indicated by symbol g in the source cell.

We use AsynchND+DS as a preliminary phase for a traversal algorithm, T, which is Echo or a distributed DFS algorithm and starts only when g is available in the source cell. The complete xP solution, denoted as AsynchND+DS+T, is constructed by the sequential composition, $\Pi_1; \Pi_2$ (§ 2.3.7), where Π_2 starts only after Π_1 terminates.

1. Π_1: AsynchND+DS;
2. Π_2: a modified version of the traversal algorithm, T (Echo or a distributed DFS algorithm), where symbol g is added as one left-side symbol of rule 1.1 that starts the computation of Π_2, shown as follows.

Echo:	$S_2\ g\ \rightarrow_{\texttt{min.min}}\ S_3\ f_i \mid \iota_i\ s \neg p_i$
Classical DFS:	$S_2\ g\ \rightarrow_{\texttt{min.min}}\ S_2\ f_i \mid \iota_i\ s \neg v$
Awerbuch's DFS:	$S_2\ g\ \rightarrow_{\texttt{min.min}}\ S_2\ f_i \mid \iota_i\ s \neg v$
Cidon's DFS:	$S_2\ g\ \rightarrow_{\texttt{min.min}}\ S_2\ f_i \mid \iota_i\ s \neg v$
Sharma et al.'s DFS:	$S_2\ g\ \rightarrow_{\texttt{min.min}}\ S_2\ f_i \mid \iota_i\ s \neg v$
Makki et al.'s DFS:	$S_2\ g\ \rightarrow_{\texttt{min.min}}\ S_2\ f_i\ y_i \mid \iota_i\ s \neg v$

We also use AsynchND+DS as a preliminary phase for AsynchBFS+DS. The control layer of the Dijkstra-Scholten algorithm can be shared by both AsynchND and AsynchBFS. Thus, the complete solution, denoted as AsynchND+AsynchBFS+DS, is constructed by sequential composition and parallel composition with interaction, $(\Pi_1; \Pi_2) \triangleright \Pi_3$ (§ 2.3.7).

1. Π_1: Π_1^* of AsynchND+DS (modified AsynchND);
2. Π_2: Π_1^* of AsynchBFS+DS (modified AsynchBFS), with an additional change to rule 1.1, where symbol g is added as one left-side symbol :

$$S_2\ g\ \rightarrow_{\texttt{min.min}} S_2\ f_i\ u_i(\lambda) \mid \iota_i\ s \neg v;$$

3. Π_3: Π_2 of AsynchND+DS or Π_2 of AsynchBFS+DS (the control layer of the Dijkstra-Scholten algorithm).

3.8 Summary

This chapter develops fundamental traversal algorithms and solves the termination detection problems in xP systems, which further validates our xP systems as a unified model for both synchronous and asynchronous systems.

Table 3.30 compares the runtime complexities of our xP specifications and the corresponding distributed algorithms. Specifically, all DFS xP specifications take one more time unit for the source cell to clean up useless symbols and this one time unit is omitted in the complexity analysis. The runtime of a termination detection algorithm is the detection latency (+) of the partial-explicit terminating algorithm, which may change when adapted to the specific basic algorithm. As shown, the runtime complexities of xP specifications are the *same* as the corresponding distributed algorithms [62].

Table 3.30: Comparisons of traversal algorithms and termination detection algorithms in terms of empirical and theoretical runtime complexities, message complexities and topology knowledge requirements, where diam is the diameter of graph $G = (V, E)$, $n = |V|$, $m = |E|$, $0 \leq r < 1$, the value of r depends on the topology [43], M is the number of messages of the basic algorithm and (+) indicates the detection latency.

	Algorithm	Time units	Time complexity	Message complexity	Topology
	Echo	3	$O(n)$	$2m$	neighbour IDs
DFS	Classical	18	$2m$	$2m$	neighbour IDs
	Awerbuch	22	$4n-2$	$4m$	neighbour IDs
	Cidon	10	$2n-2$	$3m$	neighbour IDs
	Sharma	10	$2n-2$	$2n-2$	neighbour IDs
	Makki	9	$(1+r)n$	$(1+r)n$	neighbour IDs
BFS	SynchBFS	2	O(diam)	$O(m)$	neighbour IDs
	AsynchBFS	2	O(diam)	O(diam m)	neighbour IDs
ND	SynchND	3	O(diam)	$O(m)$	
	AsynchND	1	O(diam)	$O(m)$	
TD	Dijkstra-Feijen-Van Gasteren (adapted to SynchBFS)	+7	$O(n)$	$O(n)$	ring successor ID
	Safra (adapted to AsynchBFS)	+7	$O(n)$	unbounded	ring successor ID
	Dijkstra-Scholten (adapted to AsynchBFS)	+4	$O(n)$	$O(M)$	neighbour IDs or neighbours' count

Table 3.31 compares our xP system sizes (§1.2) with the pseudocode sizes (§1.1) in Tel's book [62]. The xP system size of a termination detection algorithm include the rules of the termination detection algorithm and the rules corresponding to additions to the basic algorithm, which may change when adapted to the specific basic algorithm. For all traversal algorithms, the xP system sizes are approximately *half* of the pseudocode sizes. For the termination algorithms, xP system sizes of the DFG

and Dijkstra-Scholten algorithms are approximately 1/3 of the pseudocode sizes; the xP system size of Safra's algorithm is 3/4 of the pseudocode size.

Table 3.31: Comparisons of xP system sizes and pseudocode sizes of traversal and termination detection algorithms.

	Algorithm	xP system size	Pseudocode size
	Echo	7	15
DFS	Classical	10	22
	Awerbuch	13	30
	Cidon	15	38
	Sharma	12	15
	Makki	17	33
BFS	SynchBFS	5	n/a
	AsynchBFS	14	27
ND	SynchND	3	n/a
	AsynchND	2	n/a
TD	Dijkstra-Feijen-Van Gasteren (adapted to SynchBFS)	8	22
	Safra (adapted to AsynchBFS)	20	27
	Dijkstra-Scholten (adapted to AsynchBFS)	7	27

Taking into account that they offer complete specifications, xP specifications compare favourably, both in terms of program size and runtime complexity, with high-level pseudocodes, which ignore many critical details.

Chapter 4

Disjoint Paths Problem

The disjoint paths problem finds a maximum cardinality set of edge- and node-disjoint paths, between a source node and a target node in a *digraph*. Alternative paths between two nodes are important in many fields. They are fundamental in biological remodelling, e.g., of nervous or vascular systems [21]. A number of VLSI layout models are based on the construction of node-disjoint paths between terminals [1]. Multipath routing provides effective bandwidth in networks [69] and they are used in voice over IP and video streaming, which require fast failure recovery mechanisms [41]. Disjoint paths are sought in streaming multi-core applications to avoid sharing communication links between processors [60]. The maximum matching problem in a bipartite graph can be transformed to the disjoint paths problem [16]. 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 [42].

This chapter focuses on a series of distributed synchronous depth-first search (DFS) and breadth-first search (BFS) based algorithms, to solve edge- and node-disjoint paths problems. Consider a digraph with n cells and m arcs, where f is the maximum number of disjoint paths and d is the outdegree of the source node. We first review the classical algorithm based on Ford-Fulkerson's maximum flow algorithm [24], which runs in $O(mf)$. Then we describe Dinneen et al.'s [18] proposal based on the classical DFS, here called DFS-Edge-A* and DFS-Node-A*, which respectively solve edge- and node-disjoint paths problems in $O(mf)$. Next, we present our joint work: (1) using Cidon's DFS and a result, Theorem 4.5, Nicolescu and Wu [52] proposed improved DFS-based edge- and node-disjoint paths algorithms, here called DFS-Edge-B and DFS-Node-B, both of which run in $O(nf)$; (2) using a different idea, ElGindy, Nicolescu and Wu [22] proposed DFS-based algorithms, here called DFS-Edge-C and DFS-Edge-D, which run in $O(nd)$ and $O(nf)$, respectively; (3) Nicolescu and Wu [51, 52] also presented two BFS-based algorithms, here called BFS-Edge-A and BFS-Node-A, both of which run in $O(nf)$. Finally, my own work [65] is presented, here called BFS-Edge-B, which is a slightly improved BFS-based algorithm that also runs in $O(nf)$.

All these algorithms have been inspired and guided by our modelling approaches, but are suitable for any distributed implementation. Such modelling approaches allow us to assess the merits and suitability of xP systems as directly executable formal specifications of distributed algorithms. These algorithms are also experimentally benchmarked using their xP specifications.

This chapter is organised as follows. Section 4.1 explains the edge- and node-disjoint paths problems in digraphs. Section 4.2 describes the preliminary phase, which builds neighbourhood knowledge in a digraph [22], and it is the common phase for all xP specifications in this chapter. Sections 4.3 and 4.4 discuss DFS-based algorithms for edge- and node-disjoint paths problems. Sections 4.5 and 4.6 discuss BFS-based algorithms for edge- and node-disjoint paths problems. Section 4.7 shows the empirical performance of our algorithms. Finally, Section 4.8 summarises this chapter and compares our xP specifications with the corresponding distributed algorithms, in terms of time complexity and program size.

4.1 Disjoint Paths

This section firstly describes edge- and node-disjoint paths problems in digraphs, then discusses the special technique used to simulate node splitting in P systems for the node-disjoint paths problem, and later presents the uniform design of all our xP solutions for disjoint paths problems.

4.1.1 Disjoint Paths in Digraphs

Consider a *digraph*, $G = (V, E)$, where V is a finite set of *nodes* (cells), $V = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, and E is a set of *arcs*.

Digraph arcs define (*parent*, *child*) relationships, e.g., arc $(\sigma_i, \sigma_j) \in E$ defines σ_j as σ_i's child and σ_i as σ_j's parent; with alternate notations, $\sigma_j \in E(\sigma_i)$, $\sigma_i \in E^{-1}(\sigma_j)$. Arcs (σ_i, σ_j) and $(\sigma_j, \sigma_i) \in E$ are *symmetric*. 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 $\overline{\pi}^{-1} = \{(\sigma_j, \sigma_i) \mid (\sigma_i, \sigma_j) \in \overline{\pi}\} \subseteq E^{-1}$ as its reversal.

Given a *source* node, $s \in V$, and a *target* node, $t \in V$, 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*, it can be always streamlined at the end. The edge-disjoint paths problem can be transformed to a *maximum flow* problem by assigning *unit capacity* to each edge. Cormen et al. give a detailed presentation of these topics [16].

Given a set of edge- or node-disjoint paths P, we denote the number of edge- or node-disjoint paths as $|P|$; we define $\overline{P}$ as the set of their arcs, $\overline{P} = \cup_{\pi \in P} \overline{\pi}$; and we define 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 $\overline{P}$. In order to differentiate G with G_p, G is also called the *original* digraph.

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 (1) is known as a *forward* operation and case (2) is known as a *push-back* operation. When case (2) occurs, the arc in $\overline{P}$, (σ_i, σ_j), and its reversal in $\overline{\alpha}$, (σ_j, σ_i), are called *cancelling arcs*, which "cancel" each other (due to the 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 [24].

Example 4.1. Figure 4.1 shows how to find an augmenting path in a residual digraph [52]:

(a) shows the original digraph, G, with two edge-disjoint paths, $P = \{\sigma_1.\sigma_2.\sigma_5.\sigma_8, \sigma_1.\sigma_3.\sigma_6.\sigma_8\}$;

(b) shows the residual digraph, G_P, formed by reversing disjoint path arcs;

(c) shows an augmenting path in G_P, $\alpha = \sigma_1.\sigma_4.\sigma_6.\sigma_3.\sigma_7.\sigma_8$, which uses a reversed path arc, (σ_6, σ_3);

(d) discards the cancelling arcs, (σ_3, σ_6) and (σ_6, σ_3);

(e) relinks the remaining path fragments, $\sigma_1.\sigma_2.\sigma_5.\sigma_8$, $\sigma_1.\sigma_3$, $\sigma_6.\sigma_8$, $\sigma_1.\sigma_4.\sigma_6$ and $\sigma_3.\sigma_7.\sigma_8$, resulting in a larger set of three edge-disjoint paths, $P' = \{\sigma_1.\sigma_2.\sigma_5.\sigma_8, \sigma_1.\sigma_3.\sigma_7.\sigma_8, \sigma_1.\sigma_4.\sigma_6.\sigma_8\}$;

(f) shows the new residual digraph, $G_{P'}$.

Augmenting paths can be searched in a residual digraph using a DFS algorithm [24] or a BFS algorithm [20], which dynamically builds *search trees*: *DFS trees* or *BFS trees*, respectively. Conceptually, DFS explores as far as possible along a single branch before backtracking, while BFS explores as many branches as possible concurrently.

Intuitively, a *search tree* is built by dynamically evolving *search paths* that start from the source and try to reach the target. At any given time, a search path is, either (1) a branch in the search tree or a prefix of it or (2) a branch in the search tree followed by one more arc, which, in a failed attempt, visits another node of the same branch or of another branch [22].

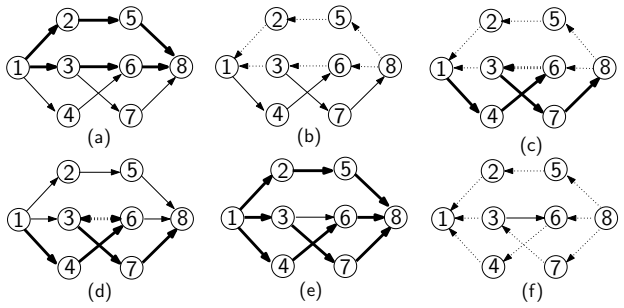


Figure 4.1: Finding an augmenting path in a residual digraph. Thin arcs: original arcs; thick arcs: disjoint or augmenting path arcs; dotted arcs: reversed path arcs.

Given a *search path* arc, (σ_i, σ_j), σ_i is a search path (sp) predecessor of σ_j, and σ_j is a search path (sp) successor of σ_i. Given a *search tree* arc, (σ_i, σ_j), σ_i is a search tree (st) predecessor of σ_j, and σ_j is a search tree (st) successor of σ_i. Given a *disjoint path* arc, (σ_i, σ_j), σ_i is a disjoint path (dp) predecessor of σ_j, and σ_j is a disjoint path (dp) successor of σ_i.

The search is *successful* when the search path reaches the target. A *successful* search path becomes a new *augmenting path* and is used to increase the number of disjoint paths: although this is conceptually a distinct operation, the new disjoint paths are typically formed while the successful search path returns on its steps back to the source. Conceptually, this solves the edge-disjoint paths problem.

However, the *node-disjoint* paths solution requires additional refinements—usually by *node splitting* [39]. Each *intermediate* node, σ_i, is split into an *entry* node, σ_i', and an *exit* node, σ_i'', linked by an arc (σ_i', σ_i''). Arcs that were directed into σ_i are redirected into σ_i' and arcs that were directed out of σ_i are redirected out of σ_i''. Figure 4.2 illustrates this node splitting, where all intermediate nodes are split.

4.1.2 Simulation of Node Splitting

As mentioned, the *node-disjoint* problem requires special treatment, e.g., node splitting. 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. [18], rather than actually splitting P cells, node splitting is simulated by

1. constraining in- and out-flow capacities of each intermediate cell to one and
2. having two *visited* marks for each intermediate cell, σ_i, one for a *virtual entry* node, σ_i', and the other one for a *virtual exit* node, σ_i''.

This approach extends the visiting idea of classical search algorithms and can be used in other distributed networks, where nodes cannot be split.

Each virtual node can be individually visited at most once in a search round. However, the virtual entry and exit 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: a search path must be prohibited to visit a cell's entry node after visiting the cell's exit node. There is no restriction, if the search path starts by visiting the entry node [52].

Now we explain the visiting restriction. 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 a reversed path arc, e.g., (σ_j', σ_i''), 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 reversed path 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, $\sigma_i.\sigma_k.\ldots.\sigma_i$, will occur [52]. Example 4.2 shows such an example, where $i = 2, j = 3, k = 4$, assuming a hypothetical arc (σ_4, σ_2).

Example 4.2. Figure 4.2 illustrates a scenario when one cell, σ_3, is visited *twice*, first via its entry and then via its exit node [18]. Assume that we already have a disjoint path, $\pi = \sigma_1.\sigma_2.\sigma_3.\sigma_4.\sigma_5$. Consider the residual digraph, $G_P = (V, E_P)$, where $P = \{\pi\}$.

A search path, τ, starts from the source cell, σ_1, and reaches σ_3, in fact, σ_3's virtual entry node, σ_3'. This is allowed and σ_3's entry node is marked as visited. However, to constrain its in-flow to one, σ_3 can only push-back τ on a reversed path arc, (σ_3, σ_2). Cell σ_2's exit node, σ_2'', becomes visited; σ_2's out-flow becomes zero and τ continues on σ_2's outgoing arc, (σ_2, σ_4). When τ reaches σ_4, σ_4's entry node, σ_4', becomes visited and σ_4 pushes τ back on a reversed path arc, (σ_4, σ_3). Cell σ_3's exit node, σ_3'', becomes visited, σ_3's out-flow becomes zero and τ continues on σ_3's outgoing arc, (σ_3, σ_5). Finally, the search path, τ, reaches the target, σ_5, and becomes an augmenting path, $\tau = \sigma_1.\sigma_3.\sigma_2.\sigma_4.\sigma_3.\sigma_5$. After removing cancelling arcs, (σ_2, σ_3) and (σ_3, σ_2), (σ_3, σ_4) and (σ_4, σ_3), and relinking the remaining arcs, $\sigma_1.\sigma_2$, $\sigma_4.\sigma_5$, $\sigma_1.\sigma_3$, $\sigma_2.\sigma_4$ and $\sigma_3.\sigma_5$, we obtain two node-disjoint paths, $\sigma_1.\sigma_2.\sigma_4.\sigma_5$ and $\sigma_1.\sigma_3.\sigma_5$.

Note that, as required by the above visiting restriction, after taking an outgoing arc from the exit node, σ_2'', path τ is prohibited to ever visit the corresponding entry node, σ_2', e.g., via a hypothetical arc (σ_4, σ_2) [52].

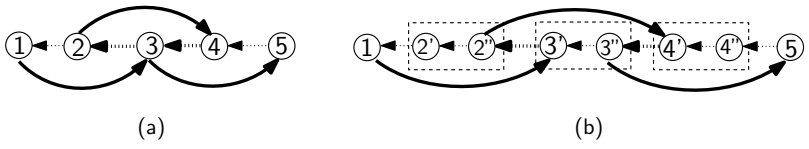


Figure 4.2: Simulating node splitting for determining node-disjoint paths [18]. Virtual entry nodes are indicated by single quotes and virtual exit nodes by double quotes.

4.1.3 Disjoint Paths in xP Systems

All our disjoint paths xP solutions share a *uniform design*, where cells start without any kind of network knowledge, i.e. cells do not know the identities of their neighbours, not even their numbers.

Each xP specification is constructed by sequential rule fragments composition (§ 2.3.7), $\Pi_1; \Pi_2$.

- Π_1, a preliminary phase, which builds topology knowledge;
- Π_2, which searches disjoint paths.

Identical or similar symbols and rulesets are used and the neighbourhood information and path information are reified as pointer symbols:

- n'_j—indicating that σ_j is a structural parent;
- n''_k—indicating that σ_k is a structural child;
- d'_j—indicating that σ_j is a disjoint path (dp) predecessor;
- d''_k—indicating that σ_k is a dp-successor;
- s'_j—indicating that σ_j is a search path (sp) predecessor;
- s''_k—indicating that σ_k is a sp-successor.

Table 4.1 shows distributed routing records for two disjoint paths of Figure 4.1 (a). Other paths, e.g., search paths, can be represented in a similar way.

Table 4.1: Distributed "routing" records for two disjoint paths of Figure 4.1 (a).

Cell	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	σ_8
Parents		n'_1	n'_1	n'_1	n'_1	n'_3 n'_4	n'_3	n'_5 n'_6 n'_7
Children	n''_2 n''_3 n''_4	n''_5	n''_6 n''_7	n''_6	n''_8	n''_8	n''_8	
Dp-predecessors		d'_1	d'_1		d'_2	d'_3		d'_5 d'_6
Dp-successors	d''_2 d''_3	d''_5	d''_6		d''_8	d''_8		

4.2 Neighbour Discovery in Digraphs

As mentioned in Section 4.1.3, in this chapter, each xP specification, $\Pi_1; \Pi_2$, is constructed by sequential composition (§ 2.3.7), where Π_1 is a preliminary neighbour discovery phase and Π_2 searches disjoint paths. This section provides a *synchronous* solution for Π_1, SynchNDD, which builds local topology knowledge in a *digraph*.

The matrix R of Π_1 consists of three vectors, informally presented in one group consisting of three subgroups, according to their functionality and applicability [22].

xP Specification 4.1: Synchronous Neighbour Discovery in Digraphs (SynchNDD)

Input: All cells start in the same state, S_0, with the same set of rules. Each cell, σ_i, contains an immutable cell ID symbol, ι_i. The source cell, σ_s, is additionally marked with one symbol, s.

Output: All cells end in the same state, S_3, and cell IDs are intact. Cell σ_s is still marked with one s. Each cell contains its structural parent pointers, n'_j's, and its structural children pointers, n''_k's.

Symbols and states

Cell σ_i sends out the following symbols:

- n is its visit token;
- n'_i is a pointer to itself, which is sent to its structural children;
- n''_i is a pointer to itself, which is sent to its structural parents.

State S_0 is an unvisited state and states S_1, S_2 and S_3 are visited states.

1. Neighbour discovery

1.1. Broadcast tokens

1.1.1. $S_0 \ \rightarrow_{\texttt{min}} \ S_1 \ n \mid s$

1.1.2. $S_0 \ n \ \rightarrow_{\texttt{min.min}} \ S_1 \ (n)\updownarrow \ (n''_i)\uparrow \ (n'_i)\downarrow \ \mid \iota_i$

1.2. Wait one step

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

1.3. Remove superfluous tokens

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

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

Initial and final configurations

Table 4.2 shows the initial and final configurations of xP Specification 4.1 for Figure 4.1.

Partial traces

Table 4.3 shows the partial traces (evolutions) of SynchNDD for Figure 4.1.

Table 4.2: Initial and final configurations of SynchNDD for Figure 4.1.

Step	σ_1	σ_2	σ_3	σ_4
0	$S_0\ \iota_1\ s$	$S_0\ \iota_2$	$S_0\ \iota_3$	$S_0\ \iota_4$
6	$S_3\ \iota_1\ s\ n''_2\ n''_3\ n''_4$	$S_3\ \iota_2\ n'_1\ n''_5$	$S_3\ \iota_3\ n'_1\ n''_6\ n''_7$	$S_3\ \iota_4\ n'_1\ n''_6$

Step	σ_5	σ_6	σ_7	σ_8
0	$S_0\ \iota_5$	$S_0\ \iota_6$	$S_0\ \iota_7$	$S_0\ \iota_8\ t$
6	$S_3\ \iota_5\ n'_2\ n''_8$	$S_3\ \iota_6\ n'_3\ n'_4\ n''_8$	$S_3\ \iota_7\ n'_3\ n''_8$	$S_3\ \iota_8\ t\ n'_5\ n'_6\ n'_7$

Rule explanations

The source cell, σ_s, generates one symbol, n, which simulates that σ_s receives a visit token from a non-existing cell (rule 1.1.1).

When unvisited cell σ_i (specifically at the start, $\sigma_i = \sigma_s$) receives one or more n's, it sends a symbol encoding its own cell ID, n'_i to its children, n''_i to its parents, and broadcasts n to all its neighbours (rule 1.1.2). Two steps after being visited (rule 1.2.1), σ_i discards the accumulated copies of n (rules 1.3.1–2).

4.3 DFS-based Edge-disjoint Paths Algorithms

This section presents a series of DFS-based edge-disjoint paths algorithms.

- DFS-Edge-A is a distributed version of the classical edge-disjoint paths algorithm, based on Ford-Fulkerson's maximum flow algorithm [24] and the classical distributed DFS (§ 3.3.2).
- DFS-Edge-A* proposed by Dinneen et al. [18] slightly improved DFS-Edge-A by limiting the probing of the source cell's children.
- DFS-Edge-B proposed by Nicolescu and Wu in our joint work [52] improves DFS-Edge-A* by using (a) a *synchronous* version of Cidon's DFS (§ 3.3.4), which *avoids revisiting* cells previously visited in the same round, and (b) an idea based on Theorem 4.5, which detects "dead" cells, i.e. visited cells that can be safely ignored in any subsequent search, at the end of *failed* rounds. These "dead" cells are discarded as fast as possible, on *shortest paths* by broadcasts.
- DFS-Edge-B* proposed by Nicolescu and Wu in our joint work [52] is a modified version of DFS-Edge-B, which is used for performance tests in this thesis. It uses the synchronous version of Cidon's DFS but intentionally *omits* to discard "dead" cells in *failed* rounds.
- DFS-Edge-C proposed by ElGindy, Nicolescu and Wu in our joint work [22] also uses a synchronous version of Cidon's distributed DFS (§ 3.3.4) but a *different* idea to discard "dead" cells identified during *successful* and *failed* rounds. These

Table 4.3: Partial traces of SynchNDD for Figure 4.1.

Step	σ_1	σ_2	σ_3	σ_4
0	$\Rightarrow n$			
1	$n \Rightarrow$ $\{n\ n'_1\}_{2,3,4}$			
2		$\{n\} \Rightarrow$ $\{n\ n''_2\}_1\ \{n\ n'_2\}_5$	$\{n\} \Rightarrow$ $\{n\ n''_3\}_1\ \{n\ n'_3\}_{6,7}$	$\{n\} \Rightarrow$ $\{n\ n''_4\}_1\ \{n\ n'_4\}_6$
3	$\{n^3\} \Rightarrow$			
4		$\{n\} \Rightarrow$	$\{n^2\} \Rightarrow$	$\{n\} \Rightarrow$
5				
6				

Step	σ_5	σ_6	σ_7	σ_8
0				
1				
2				
3	$\{n\} \Rightarrow$ $\{n\ n''_5\}_2\ \{n\ n'_5\}_8$	$\{n^2\} \Rightarrow n$ $\{n\ n''_6\}_{3,4}\ \{n\ n'_6\}_8$	$\{n\} \Rightarrow$ $\{n\ n''_7\}_3\ \{n\ n'_7\}_8$	
4				$\{n^3\} \Rightarrow n^2$ $\{n\ n'_8\}_{5,6,7}$
5	$\{n\} \Rightarrow$	$\{n\}\ n \Rightarrow$	$\{n\} \Rightarrow$	
6				$n^2 \Rightarrow$

"dead" cells are discarded on the current *search path trace*, which is typically longer than the shortest path possible (especially in digraphs).

- DFS-Edge-C* proposed by ElGindy, Nicolescu and Wu in our joint work [22] is a *restricted* version of DFS-Edge-C, which is used for performance tests in this thesis. It intentionally *omits* to discard "dead" cells found in *failed* rounds but still discards "dead" cells identified in *successful* rounds. It is used to better assess the power of the main idea behind DFS-Edge-C: even its restricted version still detects a superset of all "dead" cells detected by DFS-Edge-B.

  Note that, due to digraphs propagation delays, DFS-Edge-C and C* are not always able to prune all detected cells in as fast as possible: they could prune all, if allowed to run longer.

- DFS-Edge-D proposed by ElGindy, Nicolescu and Wu in our joint work [22] combines the benefits of DFS-Edge-B and C, which detects "dead" cells that can be identified by both algorithms. As mentioned before, the "dead" cells detected by the idea of DFS-Edge-B are discarded as fast as possible but the "dead" cells detected by the idea of DFS-Edge-C are not always discarded as fast as possible.

Table 4.4 compares the algorithms discussed above, where column "successful rounds" and "failed rounds" indicate whether the algorithm detects "dead cell" in successful and failed rounds. Note that, $\surd\times$ for "Fastest discarding time" of DFS-Edge-D indicates that DFS-Edge-D combines the benefits of DFS-Edge-B and C in the discarding process, as mentioned before.

Table 4.4: Comparisons of DFS-based edge-disjoint paths algorithms.

	Algorithm	DFS	Discard cells in successful rounds	Discard cells in failed rounds	Fastest discarding time
Previous algorithms	DFS-Edge-A	Classical DFS	×	×	×
	DFS-Edge-A*	Classical DFS	×	×	×
Our proposals	DFS-Edge-B	Cidon DFS	×	√	√
	DFS-Edge-B*	Cidon DFS	×	×	×
	DFS-Edge-C	Cidon DFS	√	√	×
	DFS-Edge-C*	Cidon DFS	√	×	×
	DFS-Edge-D	Cidon DFS	√	√	√×

Algorithms DFS-Edge-B, C, C* and D share the following features to implement their optimisations.

- Cells and arcs transit through the following three visited states: *unvisited*, *temporarily visited* and *permanently visited*.
- Search rounds explore *unvisited* cells and arcs.
- Cells and arcs first traversed during the search are marked as *temporarily visited*.
- Temporarily visited cells and arcs can be detected "dead" and marked as *permanently visited* (logically discarded), which will be ignored in any subsequent search.
- At the end of each successful round, *temporarily visited* cells and arcs that are not detected "dead" are reset to the *unvisited* status and can be revisited by next search round.

Except DFS-Edge-B* and C* that are only used for performance tests, we describe all our other algorithms using high-level pseudocodes and discuss their correctness. To improve the readability, the pseudocodes are presented in *sequentialised* versions or *structural parallel* versions (§ 1.1), which use structural and virtual arcs between cells. Then we present our xP specifications, in which parent and child cells record their corresponding arc endpoints, building a simple form of distributed routing tables (such as used in networking).

The xP specifications have challenging tasks:

- to formalise, in a fully *distributed* way, the informal description given by the high-level *sequentialised* or *structural parallel* pseudocodes, completing all important details ignored by the pseudocodes, without increasing the runtime complexity;
- to indicate
  - how to build local digraph neighbourhood awareness,
  - how to build and navigate over virtual residual digraphs,
  - how to transform augmenting paths into disjoint paths,
  - how to discard "dead" cells, and
  - how to manage concurrent notification processes.

4.3.1 DFS Token Handling Rules in Residual Digraphs

DFS-Edge-A, B, B*, C, C* and D use DFS to reach the target in a *virtual residual digraph*, by one *single token* that plays two different roles: *forward* and *backtrack*. We call the token according to its role:

- *forward token*: the token in the forward role (the forward token can be in the forward or push-back mode, § 4.1.1);
- *backtrack token*: the token in the backtrack role.

Note that, the search is based on a virtual residual digraph, where some residual arcs are reversed original arcs, some residual parents are original children and some residual children are original parents. The actual algorithm needs additional housekeeping to properly send the forward token (1) in the forward mode over an original arc to an original child, which is neither a disjoint path (dp) successor (the original arc must not be a path arc) nor a dp-predecessor (explained below), or (2) in the push-back mode over a reversed original arc (reversed path arc) to a dp-predecessor.

Figure 4.3 shows a scenario where a cell's original child is also its dp-predecessor. In the original digraph, arcs (σ_2, σ_3) and (σ_3, σ_2) are symmetric. Assume that we already have a disjoint path, $\sigma_1.\sigma_2.\sigma_3.\sigma_4.\sigma_5$, so σ_3's original child, σ_2, is also σ_3's dp-predecessor. Consider a search path, τ, which starts from σ_1 and reaches σ_3. It should push-back to σ_2 over a reverse arc (dotted arc) as discussed above in case (2), instead of reaching σ_2 over an original arc (thin arc) that is prohibited in case (1). Finally, we obtain $\sigma_1.\sigma_2.\sigma_5$ and $\sigma_1.\sigma_3.\sigma_4.\sigma_5$, therefore avoiding symmetric arcs in the disjoint paths.

All of DFS-Edge-B, B*, C, C* and D use Cidon's DFS to avoid revisiting an already visited cell and thus an *intermediate* cell becomes a *frontier* cell on receiving a forward or backtrack token. An *intermediate* cell handles a received forward or

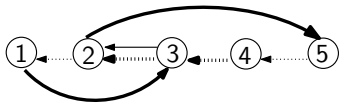


Figure 4.3: A scenario that a cell's original child is also its dp-predecessor. Thin arcs: original arcs; thick arcs: search path arcs; dotted arcs: reverse arcs (reversed path arcs).

backtrack token based on the following *token handling rules*, which are similar to the token handling rules in Section 3.3.1, except that these rules are adapted to work on a virtual residual digraph using Cidon's DFS.

(I) When an *unvisited* intermediate cell, σ_i, receives a *forward* token from its parent (in the forward mode) or its dp-successor (in the push-back mode), σ_j, cell σ_i (i) records its sp-predecessor as σ_j, (ii) marks itself as *visited* and (iii) sends its visited notifications to all its parents, becoming a *frontier* cell.

(II) When a *visited* intermediate cell, σ_i, receives a *backtrack* token from its sp-successor, σ_k, cell σ_i erases its sp-succesor pointer to σ_k, becoming a *frontier* cell.

As a frontier, σ_i sends its forward token (in the forward or push-back mode) or backtrack token as follows.

(a') if σ_i has any unvisited child, σ_k, which is neither its dp-predecessor nor its dp-successor, it sends its *forward* token (in the forward mode) to σ_k and records its sp-successor as σ_k;

(a") otherwise, if σ_i has any unvisited dp-predecessor, σ_k, it sends its *forward* token (in the push-back mode) to σ_k and records its sp-successor as σ_k;

(b) otherwise, σ_i sends its *backtrack* token to its sp-predecessor, σ_j, and erases its sp-predecessor pointer to σ_j.

4.3.2 Algorithm DFS-Edge-A

Algorithm DFS-Edge-A is a distributed version of the classical Ford-Fulkerson based edge-disjoint paths algorithm [24]. To find augmenting paths, it uses the classical DFS algorithm (§ 3.3.2). Pseudocode 4.1 shows its high-level description after unrolling its first DFS call.

DFS-Edge-A is described by *sequentialised* Pseudocodes 4.1 and 4.2, which use the following *global* variables: $G = (V, E)$ is the digraph; $\sigma_s \in V$ is the source cell; $\sigma_t \in V$

is the target cell; r is the current round number; P_{r-1} is the set of edge-disjoint paths available at the start of round r, $P_0 = \emptyset$; $G_{r-1} = (V, E_{r-1})$ is the residual digraph available at the start of round r, $G_0 = (V, E_0) = (V, E) = G$. DFS-Edge-A starts with an empty set of edge-disjoint paths, $P_0 = \emptyset$, and the trivial residual graph, $G_0 = (V, E_0)$, where $E_0 = E$ (i.e. $G_0 = G$).

Calls to DFS, lines 4.1.10 and 4.2.9, represent forward token operations. A null return from DFS represents a backtrack token operation. Line 4.2.12 represents a backtrack after systematically probing all children cells. Line 4.2.5 represents another type of backtrack: it occurs immediately after probing an already visited cell, which frequently happens in the classical DFS.

Pseudocode 4.1: DFS-Edge-A

4.1.1 **Input** : a digraph $G = (V, E)$, a source cell, $\sigma_s \in V$,
4.1.2 and a target cell, $\sigma_t \in V$
4.1.3 $r = 0, P_0 = \emptyset$, $G_0 = G$
4.1.4 **repeat**
4.1.5 $\alpha =$ **null**
4.1.6 $\beta =$ **null**
4.1.7 **while** there is an *unvisited* arc $(\sigma_s, \sigma_q) \in E_{r-1}$ **and** $\beta =$ **null**
4.1.8 $r = r + 1$
4.1.9 **set** σ_s and (σ_s, σ_q) as *visited*
4.1.10 $\beta = \text{DFS}(\sigma_q, \sigma_t, G_{r-1})$ // Pseudocode 4.2
4.1.11 **if** $\beta =$ **null then** // failed round
4.1.12 $G_r = G_{r-1}$
4.1.13 **endif**
4.1.14 **endwhile**
4.1.15 **if** $\beta \neq$ **null then** // successful round
4.1.16 $\alpha = \sigma_s.\beta$
4.1.17 $\overline{P_r} = (\overline{P_{r-1}} \setminus \overline{\alpha}^{-1}) \cup (\overline{\alpha} \setminus \overline{P_{r-1}}^{-1})$
4.1.18 $G_r = (V, E_r)$, where $E_r = (E \setminus \overline{P_r}) \cup \overline{P_r}^{-1}$
4.1.19 **reset** all *visited* cells and arcs to *unvisited*
4.1.20 **endif**
4.1.21 **until** $\alpha =$ **null**
4.1.22 **Output** : P_r, which is a maximum cardinality set of edge-disjoint paths

Pseudocode 4.2: Classical DFS, adapted for $\mathbf{G_{r-1} = (V, E_{r-1})}$

4.2.1 $\text{DFS}(\sigma_i, \sigma_t, G_{r-1})$
4.2.2 **Input** : the current cell, $\sigma_i \in V$; the target cell, $\sigma_t \in V$;
4.2.3 the current residual digraph, G_{r-1}
4.2.4 **if** $\sigma_i = \sigma_t$ **then return** σ_t // success
4.2.5 **if** σ_i is *visited* **then return null** // backtrack token
4.2.6 **set** σ_i as *visited*
4.2.7 **foreach** *unvisited* $(\sigma_i, \sigma_k) \in E_{r-1}$

4.2.8 **set** (σ_i, σ_k) as *visited*
4.2.9 $\beta =$ DFS$(\sigma_k, \sigma_t, G_{r-1})$ // forward token
4.2.10 **if** $\beta \neq$ **null return** $\sigma_i.\beta$ // success chain
4.2.11 **endfor**
4.2.12 **return null** // backtrack token
4.2.13 **Output** : a σ_i-to-σ_t path, if any; otherwise, **null**

This algorithm uses a **repeat-until** loop, repeatedly probing all unvisited children (both previously unvisited children and previously visited but failed children), resetting all visited cells and arcs as unvisited after each new augmenting path, until no more augmenting paths are found.

4.3.3 Algorithm DFS-Edge-A*

Dinneen et al. proposed algorithm DFS-Edge-A* [18], which is an improved version of DFS-Edge-A. This algorithm is described by *sequentialised* Pseudocodes 4.2–4.3 and uses the same global variables as DFS-Edge-A. Additionally, variable d is the outdegree of σ_s. Without loss of generality, we assume that σ_s's children are represented by the set $\{\sigma_{s1}, \sigma_{s2}, \ldots, \sigma_{sd}\}$.

To improve the performance of DFS-Edge-A, DFS-Edge-A* works in d successive *search rounds*, defined by successive *iterations* of its **for** loop, over the source cell's outgoing arcs (line 4.3.4). Also, DFS-Edge-A* resets visited cells (to unvisited) only after a *successful* round (line 4.3.15). These ideas seem part of algorithm folklore, but Dinneen et al. have not been able to track them to a formal source.

Pseudocode 4.3: DFS-Edge-A*

4.3.1 **Input** : a digraph $G = (V, E)$; a source cell, $\sigma_s \in V$;
4.3.2 a target cell, $\sigma_t \in V$
4.3.3 $r = 0, P_0 = \emptyset$, $G_0 = G$
4.3.4 **for** $r = 1$ **to** d
4.3.5 $\alpha =$ **null**
4.3.6 $\beta =$ **null**
4.3.7 **set** σ_s and (σ_s, σ_{sr}) as *visited*
4.3.8 $\beta =$ DFS$(\sigma_{sr}, \sigma_t, G_{r-1})$ // forward token, Pseudocode 4.2
4.3.9 **if** $\beta =$ **null then** // failed round
4.3.10 $G_r = G_{r-1}$
4.3.11 **else** // successful round
4.3.12 $\alpha = \sigma_s.\beta$
4.3.13 $\overline{P_r} = (\overline{P_{r-1}} \setminus \overline{\alpha}^{-1}) \cup (\overline{\alpha} \setminus \overline{P_{r-1}}^{-1})$
4.3.14 $G_r = (V, E_r)$, where $E_r = (E \setminus \overline{P_r}) \cup \overline{P_r}^{-1}$
4.3.15 **reset** all *visited* cells and arcs to *unvisited*
4.3.16 **endif**

4.3.17 **endfor**
4.3.18 **Output** : P_r, which is a maximum cardinality set of edge-disjoint paths

DFS-Edge-A* works in successive search rounds.

Path search: Search round r attempts to find a new augmenting path, starting with $\sigma_s.\sigma_{sr}$, using the classical DFS, Pseudocode 4.2, on the current residual digraph, $G_{r-1} = (V, E_{r-1})$. Search round r starts when the source cell, σ_s, sends its *forward token* to σ_{sr} (lines 4.3.7–8).

An *unvisited* intermediate cell accepts the forward token, marks itself as visited and becomes the new frontier (line 4.2.6). A previously *visited* intermediate cell (line 4.2.5) or a frontier cell which does not have any (more) unvisited arcs (lines 4.2.11–12) sends back a *backtrack token*, to return the frontier to its search path predecessor. A current frontier cell sends a *forward token* over an arbitrarily selected unvisited arc (lines 4.2.7–9).

Successful round: 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 towards σ_s (lines 4.2.4, 4.2.10). 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 4.3.13–14 are actually done within line 4.2.10, during the return from a successful search, without requiring any extra step.

Failed round: If the search path cannot reach σ_t, then eventually the backtrack token arrives at the source, σ_s, and the round *fails* (line 4.3.9). A failed round does *not* change the current set of disjoint paths 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 (line 4.3.10); these visited cells and arcs will not be visited in the following rounds until they are reset to unvisited (at the end of a successful round), offering some speed improvements.

End-of-round reset: When the source cell, σ_s, receives the path confirmation, it initiates a global broadcast, carried by a *reset token*, which resets all visited cells and arcs to unvisited for the next search round. In a practical implementation, the reset process, line 4.3.15, is performed by a broadcast, running 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.

Finalisation: Although it is not explicit in Pseudocode 4.3, 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, which just informs all cells that the algorithm has terminated.

The correctness of DFS-Edge-A* is ensured by Lemmas 4.1 and 4.2 and Theorems 4.3 and 4.4 (see [18] for proofs).

Lemma 4.1. Round r of DFS-Edge-A* succeeds iff the residual digraph G_r has at least one augmenting path starting with $\sigma_s.\sigma_{sr}$.

Lemma 4.2. A successful round r of DFS-Edge-A* returns one augmenting path in the residual digraph G_r and a new disjoint path in the original digraph G, both starting with with $\sigma_s.\sigma_{sr}$.

Theorem 4.3. If round r of DFS-Edge-A* fails, then none of the residual digraphs $G_{r'}$, $r' > r$, has any augmenting path starting with $\sigma_s.\sigma_{sr}$.

Theorem 4.4. Algorithm DFS-Edge-A* finds a maximum cardinality set of edge-disjoint paths.

4.3.4 Algorithm DFS-Edge-B

Algorithm DFS-Edge-B proposed by Nicolescu and Wu in our joint work [52] extends DFS-Edge-A* with two powerful optimisations, which can be performed concurrently with the main search:

1. DFS-Edge-B replaces the classical DFS by a synchronous version of Cidon's distributed DFS (§ 3.3.4). When a cell is visited for the first time in a round, it sends *temporarily visited notifications* (Cidon's notifications) to all its parents. All parents are therefore timely notified and will *never* send a forward token to this already visited cell. The visited notification messages travel in *parallel* with the main search token, without affecting the overall performance. Essentially, this reduces the DFS complexity from $O(m)$ to $O(n)$.
2. DFS-Edge-B includes a procedure to *discard* "dead" cells. This procedure relies on Theorem 4.5, which proves that all cells visited during a *failed* round can indeed be ignored in any subsequent search.

Note that, a cell's temporarily visited notifications are sent to its parents; however, this is a *residual* digraph, where some residual arcs are reversed original arcs, some residual parents are original children and some residual children are original parents. The actual algorithm broadcasts such notifications to all neighbours.

DFS-Edge-B is described by *structural parallel* Pseudocodes 4.4 and 4.5 and uses the same variables as DFS-Edge-A*. Most of this discussion also applies to the node-disjoint version, DFS-Node-B (§ 4.4.1), which is based on DFS-Edge-B.

Pseudocode 4.4: DFS-Edge-B

4.4.1 **Input** : a digraph $G = (V, E)$; a source cell, $\sigma_s \in V$;
4.4.2 a target cell, $\sigma_t \in V$
4.4.3 $P_0 = \emptyset$, $G_0 = G$
4.4.4 **set** σ_s as *permanently visited*
4.4.5 **parallel foreach** arc $(\sigma_j, \sigma_s) \in E$ // Cidon's notifications for σ_s
4.4.6 **set** (σ_j, σ_s) as *permanently visited*
4.4.7 **endfor**
4.4.8 **for** $r = 1$ **to** d
4.4.9 $\alpha =$ **null**
4.4.10 $\beta =$ **null**
4.4.11 **if** σ_{sr} is *permanently visited* **then continue**
4.4.12 **set** (σ_s, σ_{sr}) as *permanently visited*
4.4.13 $\beta =$ Cidon_DFS$(\sigma_{sr}, \sigma_t, G_{r-1})$ // forward token, Pseudocode 4.5
4.4.14 **if** $\beta =$ **null then** // failed round
4.4.15 $G_r = G_{r-1}$
4.4.16 **set** all *temporarily visited* cells and arcs to *permanently visited*
4.4.17 **else** // successful round
4.4.18 $\alpha = \sigma_s.\beta$
4.4.19 $\overline{P_r} = (\overline{P_{r-1}} \setminus \overline{\alpha}^{-1}) \cup (\overline{\alpha} \setminus \overline{P_{r-1}}^{-1})$
4.4.20 $G_r = (V, E_r)$, where $E_r = (E \setminus \overline{P_r}) \cup \overline{P_r}^{-1}$
4.4.21 **reset** all *temporarily visited* cells and arcs to *unvisited*
4.4.22 **endif**
4.4.23 **endfor**
4.4.24 **Output** : P_r, which is a maximum cardinality set of edge-disjoint paths

Pseudocode 4.5: Cidon's DFS, adapted for $\mathbf{G_{r-1} = (V, E_{r-1})}$

4.5.1 Cidon_DFS$(\sigma_i, \sigma_t, G_{r-1})$
4.5.2 **Input** : the current cell, $\sigma_i \in V$; the target cell, $\sigma_t \in V$;
4.5.3 the current residual digraph, G_{r-1}
4.5.4 **if** $\sigma_i = \sigma_t$ **then return** σ_t // success
4.5.5 **set** σ_i as *temporarily visited*
4.5.6 **parallel foreach** *unvisited* arc $(\sigma_j, \sigma_i) \in E_{r-1}$ // Cidon's notifications
4.5.7 **set** (σ_j, σ_i) as *temporarily visited*
4.5.8 **endfor**
4.5.9 **foreach** *unvisited* arc $(\sigma_i, \sigma_k) \in E_{r-1}$
4.5.10 **set** (σ_i, σ_k) as *temporarily visited*
4.5.11 $\beta =$ Cidon_DFS$(\sigma_k, \sigma_t, G_{r-1})$ // forward token
4.5.12 **if** $\beta \neq$ **null then return** $\sigma_i.\beta$ // success chain
4.5.13 **endfor**
4.5.14 **return null** // backtrack token
4.5.15 **Output** : a σ_i-to-σ_t path, if any; otherwise, **null**

DFS-Edge-B also works in successive search rounds.

Path search: Search round r attempts to find a new augmenting path starting with $\sigma_s.\sigma_{sr}$, using Cidon's DFS (Pseudocode 4.5) on the current residual digraph. Search round r starts when the source cell, σ_s, sends its *forward token* to unvisited σ_{sr} (lines 4.4.8, 4.4.11).

An intermediate cell that receives the forward token becomes the new frontier and marks itself and its incoming arcs as *temporarily visited* by sending *temporarily visited notifications* to all its parents (lines 4.5.6–8). A current frontier cell sends a forward token over an arbitrarily selected unvisited arc (lines 4.5.9–11). A cell which does not have any (more) unvisited arcs sends a backtrack token to return the frontier to its search path predecessor (lines 4.5.13–14).

As previously mentioned, this is a virtual residual digraph. The actual algorithm broadcasts temporarily visited notifications. Also, it needs additional housekeeping to properly send the forward token over an original arc to an original child or over a reversed original arc to a disjoint path predecessor, as discussed in the token handling rules in Section 4.3.1.

Successful round: If the search path reaches the target cell, σ_t, then round r is successful and σ_t sends a path confirmation back towards σ_s (lines 4.5.4, 4.5.12). While moving towards σ_s, the confirmation reshapes the existing disjoint paths and the newly found augmenting path by discarding cancelling arcs and relinking the rest, building a larger set of disjoint paths and a new residual digraph. Thus, lines 4.4.19–20 are actually done within line 4.5.12, during the return from a successful search, without requiring any extra step.

Failed round: If the search path cannot reach σ_t, then eventually the backtrack token arrives at the source, σ_s, and the round fails (line 4.4.14). A failed round does *not* change the current set of disjoint paths or the current residual digraph (line 4.4.15).

End-of-round resets: If the source cell, σ_s, receives a path confirmation, it initiates an *after-success reset* broadcast, which resets all *temporarily visited* cells and arcs to *unvisited*. If the source cell, σ_s, receives a backtrack token, it initiates an *after-failure reset* broadcast, which resets all *temporarily visited* cells and arcs to *permanently visited*. Both end-of-round resets run two steps ahead and in parallel with the next search round, without affecting it.

Finalisation: Although it is not explicit in Pseudocode 4.4, after probing all its children, the source cell, σ_s, initiates a global broadcast, carried by a finalise token, to inform all cells that the algorithm has terminated.

Example 4.3. Figure 4.4 shows a scenario that DFS-Edge-B outperforms DFS-Edge-A*. Cell σ_1 is the source, cell σ_7 is the target and σ_1's children are σ_2, σ_5 and σ_6.

The round 1 search path, τ, extends identically in both DFS-Edge-B and DFS-Edge-A*, until τ reaches σ_4: $\tau = \sigma_1.\sigma_2.\sigma_3.\sigma_4$. At this point, τ takes different direc-

tions. In DFS-Edge-A*, τ extends to σ_2, finds that its was *visited*, backtracks to σ_4 and then backtracks all the way back to σ_1. However, in DFS-Edge-B, σ_4 was already notified that σ_2 is *temporarily visited*, so the backtrack starts immediately (without probing σ_2). In both algorithms, round 1 fails, but DFS-Edge-B is *faster*.

At the end of *failed* round 1, in DFS-Edge-A*, the *visited* cells, σ_2, σ_3 and σ_4, remain *visited*. However, DFS-Edge-B broadcasts to set these cells, σ_2, σ_3 and σ_4, to *permanently visited* (until the end of the algorithm).

The round 2 search path, τ', extends identically in both DFS-Edge-B and DFS-Edge-A* and succeeds: $\tau' = \sigma_1.\sigma_5.\sigma_7$. At the end of this *successful* round, DFS-Edge-A* broadcasts to reset all *visited* cells, σ_1, σ_5 and σ_7 of round 2 and σ_2, σ_3 and σ_4 of round 1, to *unvisited*. Similarly, DFS-Edge-B broadcasts to reset all *temporarily visited* cells, σ_1, σ_5 and σ_7, to *unvisited*. Both algorithms are equally fast in succeeding in this round 2.

The round 3 search path, τ'', extends differently in the two algorithms. In DFS-Edge-B, τ'' reaches target σ_7 in two steps: $\tau'' = \sigma_1.\sigma_6.\sigma_7$; it does not attempt to probe σ_4, which is *permanently visited*, i.e. conceptually removed from the graph. However, in DFS-Edge-A*, τ'', after reaching σ_6, probes σ_4 and extends all the way to $\tau'' = \sigma_1.\sigma_6.\sigma_4.\sigma_2.\sigma_3.\sigma_4$, before it finds the impasse, backtracks to σ_6 and finally succeeds. In both algorithms, round 3 succeeds with the same path, but DFS-Edge-B is *much faster*.

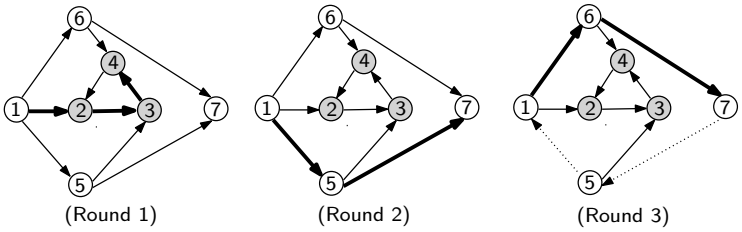


Figure 4.4: A scenario where DFS-Edge-B outperforms DFS-Edge-A*. Thin arcs: original arcs; thick arcs: search path arcs; dotted arcs: reversed path arcs; gray cells: cells discarded by DFS-Edge-B.

The correctness of algorithm DFS-Edge-B is ensured by Theorem 4.6, which follows Lemma 4.1, Lemma 4.2 and Theorem 4.5. Theorem 4.5 represents the core argument supporting the improvement of DFS-Edge-B. This core result uses a modified version of DFS-Edge-B, designated here by DFS-Edge-B*, which is obtained from DFS-Edge-B, by omitting line 4.4.16.

Essentially, DFS-Edge-B* is a faster version of DFS-Edge-A*, where cells can assume only two states, *unvisited* and (temporarily) *visited*, but never *permanently visited*. DFS-Edge-B* uses Cidon's DFS instead of the classical DFS, but this is not relevant here: these DFS algorithms differ in their runtime performance, but not in the final output.

Theorem 4.5. If round r of DFS-Edge-B* 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 DFS-Edge-B*, 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.

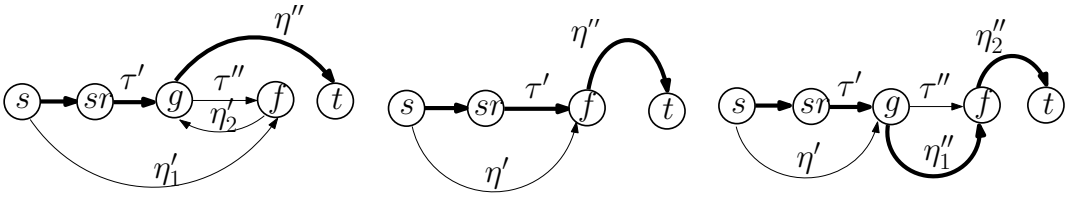
Because τ appears in the *first* round, all its arcs belong to G', $\overline{\tau} \subseteq E'$. Because α is the *first* augmenting path passing through σ_f, $\{(\sigma_{f'}, \sigma_f), (\sigma_f, \sigma_{f''})\} \subseteq E'$. Moreover, one of the new edge-disjoint paths, determined at round r'', 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 4.5 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 summarise, 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 Lemmas 4.1 and 4.2).

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

Note that Theorem 4.5 remains valid for DFS-Edge-A* (i.e. if in its statement, DFS-Edge-B* is substituted by DFS-Edge-A*). Thus, the main result supporting the



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

improvements of algorithm DFS-Edge-A*, Dinneen et al.'s Theorem 4.3 [18], is now a corollary of our Theorem 4.5.

The following result consolidates the above arguments.

Theorem 4.6. Algorithm DFS-Edge-B finds a maximum cardinality set of edge-disjoint paths.

xP Specification 4.2: DFS-Edge-B

Input: All cells start in the same initial state, S_0, with the same set of rules, without any topological awareness (they do not even know their neighbours). 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 state, S_{60}. 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: disjoint path predecessors, d'_j's, and disjoint path successors, d''_k's.

Symbols and states

Cell σ_i uses the following symbols to record its relationships with neighbour cells, σ_j and σ_k:

- n'_j indicates a structural parent;
- n''_k indicates a structural 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;
- v_j indicates a temporarily visited neighbour;
- w_j indicates a permanently visited neighbour.

Cell σ_i sends out the following symbols:

- f_i is its forward or backtrack token;
- v_i is its temporarily visited notification;
- a is a path confirmation;
- q is an after-failure reset token;
- r is an after-success reset token;
- g is a finalise token.

Also, cell σ_i uses the following symbols to record its states:

- v indicates that it is temporarily visited;
- w indicates that it is permanently visited;
- f indicates that it is the frontier cell;
- d indicates that it appears on a disjoint path.

For cell σ_i, a forward token is indicated by $f_j \neg v$, while a backtrack token is indicated by $f_j\ v$.

State S_{10} is used by the source cell; S_{20} is used by intermediate cells; S_{30} is used by the target cell. Following states are shared by all cells: S_3 is a state at the start of each round; S_{40} and S_{41} are states at the end of each round; S_{50} is a state at the end of the algorithm; S_{60} is a state after cleaning up all useless symbols.

The matrix R of xP Specification 4.2, $\Pi_1; \Pi_2$, is constructed by sequential composition of Π_1 and Π_2, where Π_1 is SynchNDD (§ 4.2) and Π_2 is the disjoint path search algorithm consisting of ten vectors, informally presented in five groups, according to their functionality and applicability.

Π_1: SynchNDD (see Section 4.2)

Π_2: Disjoint path search

2. Initial differentiation (S_3)

2.1.

2.1.1. $S_3 \ \rightarrow_{\texttt{min}} \ S_{10}\ f \mid s$

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

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

3. Source cell (S_{10})

3.1.

3.1.1. $S_{10}\ f \ \rightarrow_{\texttt{min.min}} \ S_{10}\ s''_k\ (v_i)\updownarrow \ (f_i)\downarrow_k \ \mid n''_k \neg w_k\ v_k$

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

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

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

3.1.5. $S_{10}\ v_k \ \rightarrow_{\texttt{max.min}} \ S_{40}\ w_k \mid q \neg w_k$

3.1.6. $S_{10}\ f_k \ \rightarrow_{\texttt{max.max}} \ S_{40}$

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

4. Intermediate cells (S_{20})

4.1. Finalisation

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

4.2. Frontier

4.2.1. $S_{20}\ f_j \ \rightarrow_{\texttt{min.min}} \ S_{20}\ v\ s'_j\ (v_i)\updownarrow \ f \mid \iota_i \neg w\ v$

4.2.2. $S_{20}\ f_k\ s''_k \ \rightarrow_{\texttt{min.min}} \ S_{20}\ f \mid v$

4.2.3. $S_{20}\ f_k \ \rightarrow_{\texttt{max.max}} \ S_{20}$

4.2.4. $S_{20}\ f \ \rightarrow_{\texttt{min.min}} \ S_{20}\ v_k\ s''_k\ (f_i)\downarrow_k \ \mid \iota_i\ n''_k \neg w_k\ v_k\ d'_k\ d''_k$

4.2.5. $S_{20}\ f \ \rightarrow_{\texttt{min.min}} \ S_{20}\ v_k\ s''_k\ (f_i)\uparrow_k \ \mid \iota_i\ d'_k \neg w_k\ v_k$

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

4.3. Path confirmation

4.3.1. $S_{20}\ a \ \rightarrow_{\texttt{max}} \ S_{20} \mid a$

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

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

4.4. End-of-round resets

4.4.1. $S_{20} \rightarrow_{\texttt{min}} S_{40}\ (q)\updownarrow \mid q$

4.4.2. $S_{20}\ v \rightarrow_{\texttt{min}} S_{40}\ w \mid \iota_i\ q \neg w$

4.4.3. $S_{20}\ v_k \rightarrow_{\texttt{max.min}} S_{40}\ w_k \mid q \neg w_k$

4.4.4. $S_{20} \rightarrow_{\texttt{min}} S_{40}\ (r)\updownarrow \mid r$

5. Target cell (S_{30})

5.1.

5.1.1. $S_{30}\ g \rightarrow_{\texttt{min}} S_{50}\ (g)\updownarrow$

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

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

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

6. All cells (S_{40}, S_{41}, S_{50})

6.1.

6.1.1. $S_{40} \rightarrow_{\texttt{min}} S_{41}$

6.2. End of each search round

6.2.1. $S_{41}\ v_k \rightarrow_{\texttt{max.max}} S_3$

6.2.2. $S_{41}\ v \rightarrow_{\texttt{max}} S_3$

6.2.3. $S_{41}\ a \rightarrow_{\texttt{max}} S_3$

6.2.4. $S_{41}\ q \rightarrow_{\texttt{max}} S_3$

6.2.5. $S_{41}\ r \rightarrow_{\texttt{max}} S_3$

6.2.6. $S_{41} \rightarrow_{\texttt{min}} S_3$

6.3. End of the algorithm

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

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

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

6.3.4. $S_{50}\ w_k \rightarrow_{\texttt{max.max}} S_{50}$

6.3.5. $S_{50}\ v_k \rightarrow_{\texttt{max.max}} S_{50}$

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

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

6.3.8. $S_{50}\ d \rightarrow_{\texttt{max}} S_{50}$

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

Table 4.5: Initial and final configurations of xP Specification 4.2 for Figure 4.1.

Cell	σ_1	σ_2	σ_3	σ_4
Initial	$S_0\ \iota_1\ s$	$S_0\ \iota_2$	$S_0\ \iota_3$	$S_0\ \iota_4$
Final	$S_{60}\ \iota_1\ s\ d''_2\ d''_3\ d''_4$	$S_{60}\ \iota_2\ d'_1\ d''_5$	$S_{60}\ \iota_3\ d'_1\ d''_7$	$S_{60}\ \iota_4\ d'_1\ d''_6$
Cell	σ_5	σ_6	σ_7	σ_8
Initial	$S_0\ \iota_5$	$S_0\ \iota_6$	$S_0\ \iota_7$	$S_0\ \iota_8\ t$
Final	$S_{60}\ \iota_5\ d'_2\ d''_8$	$S_{60}\ \iota_6\ d'_4\ d''_8$	$S_{60}\ \iota_7\ d'_3\ d''_8$	$S_{60}\ \iota_8\ t\ d'_5\ d'_6\ d'_7$

Initial and final configurations

Table 4.5 shows the initial and final configurations of xP Specification 4.2 for Figure 4.1.

Rule explanations

At the start of each search round, each cell enters corresponding state as follows and then searches disjoint paths.

- Rule 2.1.1: the source cell, σ_s, enters S_{10} and generates one f, becoming a frontier cell.
- Rule 2.1.2: the target cell, σ_t, enters S_{30}.
- Rule 2.1.3: an intermediate cell, σ_i, enters S_{20}.

Path search: At the start of each search round, the source cell, σ_s, sends its (forward) token, f_s, to one of its unvisited children, $n''_k \neg w_k\ v_k$, records its sp-successor as s''_k and broadcasts its temporarily visited notification, v_s, to all its neighbours (rule 3.1.1).

An intermediate cell, σ_i, handles its received (forward or backtrack) token, f_j, using the following rules corresponding to the token-handling rules in Section 4.3.1.

(I) Rule 4.2.1: when unvisited cell σ_i, $\neg\ w\ v$, receives f_j, it records s'_j, sends v_i to all neighbours and generates one f, indicating it is a frontier cell.

(II) Rule 4.2.2: when visited cell σ_i, v, receives f_k from its sp-successor, s''_k, it erases s''_k and generates one f.

As a frontier, σ_i sends its forward token (in the forward or push-back mode) or backtrack token as follows.

(a') Rule 4.2.4: if cell σ_i has an unvisited child, n''_k, which is (1) not visited in the same round, $\neg\ v_k$ (Cidon's DFS), (2) not discarded in a failed round, $\neg\ w_k$ (Theorem 4.5), and (3) neither a dp-predecessor nor a dp-successor, $\neg\ d'_k\ d''_k$, then σ_i sends f_i to σ_k and records its sp-successor as s''_k.

(a") Rule 4.2.5: otherwise, if cell σ_i has an unvisited dp-predecessor, d'_k, which is (1) not visited in the same round, $\neg\, v_k$ and (2) not discarded in a failed round, $\neg\, w_k$, then σ_i sends f_i to σ_k and records its sp-successor as s''_k.

(b) Rule 4.2.6: otherwise, cell σ_i sends f_i to its sp-predecessor, s'_j.

Successful round: On receiving a token, f_j, the target cell, σ_t, records its dp-predecessor, d'_j, and sends back a path confirmation, a, to σ_j (rule 5.1.2).

When an intermediate cell, σ_i, receives a, it transforms its sp-predecessor, s'_j, into a dp-predecessor, d'_j, and its sp-successor, s''_k, into a dp-successor, d''_k (rule 4.3.2). Cell σ_i may have already contained (from a previous round) a dp-predecessor and a sp-successor, $d'_{j'}$ and $d''_{k'}$. If $j = k'$, then d'_j and $d''_{k'}$ are deleted, as one endpoint of the cancelling arc pair, (j, i) and (i, j) (§ 4.1.1); similarly, if $k = j'$, then d''_k and $d'_{j'}$ are deleted (rule 4.3.3).

If the source cell, σ_s, receives a, it transforms its sp-successor, s''_k, into a dp-successor, d''_k, deletes child pointer n''_k, and broadcasts an after-success reset token, r (rule 3.1.2).

Failed round: If the source cell, σ_s, receives a backtrack token, f_k, it deletes s''_k and n''_k and broadcasts an after-failure reset token, q (rule 3.1.4).

End-of-round resets: Based on Theorem 4.5, at the end of a round: (1) on receiving an after-failure reset token, q, σ_i transforms v into w and and v_k's into w_k's (rules 4.4.2–3); (2) on receiving an after-success reset token, r, σ_i erases v and all v_k's (rules 6.2.1–2). After either reset, each cell enters its corresponding state again (rules 2.1.1–3).

Finalisation: Two steps after the end-of-round reset, if the source cell, σ_s, has no more unvisited children, it initiates a finalise broadcast by token g, to prompt all cells to clean up useless symbols (rule 3.1.7).

Partial traces

Table 4.6 shows cell's configurations of xP Specification 4.2 for the five snapshots in Figure 4.1. Note that the residual graph is built at the same time when the new disjoint paths are built, while confirming a new augmenting path (i.e. a successful search path).

4.3.5 Algorithm DFS-Edge-C

Algorithm DFS-Edge-C proposed by ElGindy, Nicolescu and Wu in our joint work [22] uses a different procedure to detect "dead" cells, based on two numerical search-specific *local* cell attributes: the (known) cell *depth* and a new attribute which we call *reach-number*. A cell's *depth*, σ_i.`depth`, is the number of hops from the source to itself in the search tree. A cell's *reach-number* is the minimum of its depth and all

Table 4.6: Partial traces of xP Specification 4.2 for Figure 4.1.

Fig.	σ_1	σ_2	σ_3	σ_4
Top.	$S_{10}\ \iota_1\ s\ n''_2 n''_3 n''_4$	$S_{20}\ \iota_2\ n'_1 n''_5$	$S_{20}\ \iota_3\ n'_1 n''_6 n''_7$	$S_{20}\ \iota_4\ n'_1 n''_6$
(a) (b)	$S_{10}\ \iota_1\ s\ n''_2 n''_3 n''_4$ $d''_2 d''_3$	$S_{20}\ \iota_2\ n'_1 n''_5$ $d'_1 d''_5$	$S_{20}\ \iota_3\ n'_1 n''_6 n''_7$ $d'_1 d''_6$	$S_{20}\ \iota_4\ n'_1 n''_6$
(c)	$S_{10}\ \iota_1\ s\ n''_2 n''_3 n''_4$ $d''_2 d''_3 s''_4$	$S_{20}\ \iota_2\ n'_1 n''_5$ $d'_1 d''_5$	$S_{20}\ \iota_3\ n'_1 n''_6 n''_7$ $d'_1 d''_6 s'_6 s''_7$	$S_{20}\ \iota_4\ n'_1 n''_6$ $s'_1 s''_6$
(e) (f)	$S_{10}\ \iota_1\ s\ n''_2 n''_3 n''_4$ $d''_2 d''_3 d''_4$	$S_{20}\ \iota_2\ n'_1 n''_5$ $d'_1 d''_5$	$S_{20}\ \iota_3\ n'_1 n''_6 n''_7$ $d'_1 d''_7$	$S_{20}\ \iota_4\ n'_1 n''_6$ $d'_1 d''_6$

Fig.	σ_5	σ_6	σ_7	σ_8
Top.	$S_{20}\ \iota_5\ n'_2 n''_8$	$S_{20}\ \iota_6\ n'_3 n'_4 n''_8$	$S_{20}\ \iota_7\ n'_3 n''_8$	$S_{30}\ \iota_8\ t\ n'_5 n'_6 n'_7$
(a) (b)	$S_{20}\ \iota_5\ n'_2 n''_8$ $d'_2 d''_8$	$S_{20}\ \iota_6\ n'_3 n'_4 n''_8$ $d'_3 d''_8$	$S_{20}\ \iota_7\ n'_3 n''_8$	$S_{30}\ \iota_8\ t\ n'_5 n'_6 n'_7$ $d'_5 d'_6$
(c)	$S_{20}\ \iota_5\ n'_2 n''_8$ $d'_2 d''_8$	$S_{20}\ \iota_6\ n'_3 n'_4 n''_8$ $d'_3 d''_8 s'_4 s''_3$	$S_{20}\ \iota_7\ n'_3 n''_8$ $s'_3 s''_8$	$S_{30}\ \iota_8\ t\ n'_5 n'_6 n'_7$ $d'_5 d'_6 d'_7$
(e) (f)	$S_{20}\ \iota_5\ n'_2 n''_8$ $d'_2 d''_8$	$S_{20}\ \iota_6\ n'_3 n'_4 n''_8$ $d'_4 d''_8$	$S_{20}\ \iota_7\ n'_3 n''_8$ $d'_3 d''_8$	$S_{30}\ \iota_8\ t\ n'_5 n'_6 n'_7$ $d'_5 d'_6 d'_7$

its search tree (st) successors' reach-numbers, which satisfies the following recursive equation:

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

where E' is the current residual arcs set. Moreover, we assume that *discarded* cells have *infinite* reach-numbers.

As algorithmically determined, depths and reach-numbers start as *infinite* and are further *iteratively* adjusted during the search process. The algorithm detects "dead" cells using the following *optimisation rules* and is described by *structural parallel* Pseudocodes 4.6–4.9 using the same global variables as DFS-Edge-B.

1. When the search path first *visits* cell σ_i:
   (a) σ_i marks itself and its incoming arcs as temporarily visited by sending temporarily visited notifications to all its parents (lines 4.7.6, 4.7.8–10);
   (b) σ_i's depth and reach-number are set to the current hop count (line 4.7.7).
2. When the search path *backtracks* from cell σ_k to cell σ_i:
   (a) σ_i computes its reach-number (lines 4.7.7, 4.7.13–15, 4.7.18–19), which can be *decreased* if $\sigma_k.\texttt{reach} < \sigma_i.\texttt{reach}$ (line 4.7.19);
   (b) σ_i sends a *discard notification* to σ_k, if $\sigma_k.\texttt{reach} > \sigma_i.\texttt{depth}$ (lines 4.7.20–21).
3. When cell σ_i receives a *discard notification*:

   (a) σ_i marks itself and its incoming arcs as permanently visited by sending *permanently visited notifications* to all its parents (lines 4.8.4, 4.8.6, 4.8.10);

   (b) σ_i.`reach` becomes *infinite* (line 4.8.5);

   (c) σ_i sends *update notifications* to its st-predecessors and can *increase* their reach-numbers (lines 4.8.7–9);

   (d) σ_i sends *discard notifications* to all its st-successors (this is a recursive procedure, lines 4.8.12–14).

4. When cell σ_j receives an *update notification* from σ_i, σ_j computes its reach-number (lines 4.9.3–6); if σ_j's reach-number is *increased* (because its st-successor σ_i has increased its own reach-number) (lines 4.9.7–8):

   (a) σ_j sends update notifications to its st-predecessors and can also *increase* their reach-numbers (lines 4.9.9–11);

   (b) σ_j sends a *discard notification* to σ_i, if σ_i.`reach` $> \sigma_j$.`depth` (lines 4.9.13–15).

Note that, if finite, a cell's reach-number is never greater than its own depth, i.e. σ_i.`reach` $\leq \sigma_i$.`depth`. Also note the two cases where a cell can be discarded, (2.b) and (4.b), require a similar additional condition: this cell's reach-number is changed to a finite number greater than its st-predecessor's depth.

Items (1.b), (2.a) and (3.b) can be easily incorporated in any search. However, in a message-based distributed algorithm, items (1.a) and (3.a) must be implemented by *temporarily visited notifications* and *permanently visited notifications*, respectively; items (2.b), (3.c), (3.d), (4.a) and (4.b) must be recursively propagated by notification messages over existing arcs: cases (2.b), (3.d) and (4.b) trigger *discard notifications*, which are propagated by the function Discard, and cases (3.c) and (4.a) trigger *update notifications*, propagated by the function Update.

However, this is a *virtual residual* digraph, where some residual arcs are reversed original arcs, some residual parents are original children and some residual children are original parents. The actual algorithm simply broadcasts temporarily and permanently visited notifications but needs additional housekeeping to properly send discard notifications and accept update notifications, only for concerned neighbours.

These notifications travel in *parallel* with the main search activities, without affecting the overall performance. However, the *discard and update notifications* only travel along *search paths traces*, which, in digraphs, are *not* the *shortest* possible paths. Therefore, as we see in a later example, not all cells can be effectively notified as fast as possible, and may be reached by the next search process *before* they get their due notifications. Briefly, in a digraph based system, we have a *pruning propagation delay*, which may negatively affect its performance.

At the end of a successful round, the source cell, σ_s, initiates a global broadcast, which *resets* all temporarily visited cells and arcs to unvisited (line 4.6.19). This reset

process runs two steps ahead, in parallel with the next search round without affecting it.

Pseudocode 4.6: DFS-Edge-C

4.6.1 **Input** : a digraph $G = (V, E)$, a source cell, $\sigma_s \in V$,
4.6.2 and a target cell, $\sigma_t \in V$
4.6.3 $P_0 = \emptyset$, $G_0 = G$
4.6.4 **set** σ_s as *permanently visited*
4.6.5 **foreach** *unvisited* arc $(\sigma_j, \sigma_s) \in E$
4.6.6 **set** (σ_j, σ_s) as *permanently visited*
4.6.7 **endfor**
4.6.8 **for** $r = 1$ **to** d
4.6.9 **if** σ_{sr} is *permanently visited* **then continue**
4.6.10 **set** (σ_s, σ_{sr}) as *permanently visited*
4.6.11 $\beta =$ NW_DFS$(\sigma_{sr}, \sigma_t, G_{r-1}, 1)$ // Pseudocode 4.7
4.6.12 **if** $\beta =$ **null then** // failed round
4.6.13 **fork** Discard(σ_{sr}, G_{r-1})
4.6.14 $G_r = G_{r-1}$
4.6.15 **else** // successful round
4.6.16 $\alpha = \sigma_s.\beta$
4.6.17 $\overline{P_r} = (\overline{P_{r-1}} \setminus \overline{\alpha}^{-1}) \cup (\overline{\alpha} \setminus \overline{P_{r-1}}^{-1})$
4.6.18 $G_r = (V, E_r)$, where $E_r = (E \setminus \overline{P_r}) \cup \overline{P_r}^{-1}$
4.6.19 **reset** all *temporarily visited* cells and arcs to *unvisited*
4.6.20 **endif**
4.6.21 **endfor**
4.6.22 **Output** : P_r, which is a maximum cardinality set of edge-disjoint paths

Pseudocode 4.7: NW-DFS, adapted for $\mathbf{G_{r-1} = (V, E_{r-1})}$

4.7.1 NW_DFS$(\sigma_i, \sigma_t, G_{r-1}, depth)$
4.7.2 **Input** : the current cell, $\sigma_i \in V$, the target cell, $\sigma_t \in V$,
4.7.3 the residual digraph, G_{r-1}, and σ_i's depth, *depth*
4.7.4 **if** $\sigma_i = \sigma_t$ **then return** σ_t
4.7.5 **if** σ_i is *permanently visited* **then return null**
4.7.6 **set** σ_i as *temporarily visited*
4.7.7 σ_i.reach $= \sigma_i$.depth $= depth$
4.7.8 **foreach** *unvisited* arc $(\sigma_j, \sigma_i) \in E_{r-1}$ // Cidon's DFS
4.7.9 **set** (σ_j, σ_i) as *temporarily visited*
4.7.10 **endfor**
4.7.11 **foreach** arc $(\sigma_i, \sigma_k) \in E_{r-1}$
4.7.12 **if** (σ_i, σ_k) is *permanently visited* **then continue**
4.7.13 **elseif** (σ_i, σ_k) is *temporarily visited* **then**
4.7.14 σ_i.reach $= \min(\sigma_i$.reach$, \sigma_k$.reach$)$
4.7.15 **else** // unvisited

```
4.7.16        set (σ_i, σ_k) as temporarily visited
4.7.17        β = NW_DFS(σ_k, σ_t, G_{r−1}, depth + 1)
4.7.18        if β = null then
4.7.19          σ_i.reach = min(σ_i.reach, σ_k.reach)
4.7.20          if σ_k.reach > σ_i.depth then
4.7.21            fork Discard(σ_k, G_{r−1})              // Pseudocode 4.8
4.7.22          endif
4.7.23        else
4.7.24          return σ_i.β
4.7.25        endif
4.7.26      endif
4.7.27    endfor
4.7.28    return null
4.7.29    Output: a σ_i-to-σ_t path, if any; otherwise, null
```

Pseudocode 4.8: Discard, adapted for $\mathbf{G_{r-1} = (V, E_{r-1})}$

```
4.8.1   Discard(σ_i, G_{r−1})
4.8.2   Input: a cell to discard, σ_i ∈ V, and the residual digraph, G_{r−1}
4.8.3   if σ_i is permanently visited then return
4.8.4   set σ_i as permanently visited
4.8.5   σ_i.reach = ∞
4.8.6   foreach (σ_j, σ_i) ∈ E_{r−1}
4.8.7     if (σ_j, σ_i) is temporarily visited then
4.8.8       fork Update(σ_j, G_{r−1}, σ_i)              // Pseudocode 4.9
4.8.9     endif
4.8.10    set (σ_j, σ_i) as permanently visited
4.8.11  endfor
4.8.12  foreach temporarily visited arc (σ_i, σ_k)
4.8.13    fork Discard(σ_k, G_{r−1})
4.8.14  endfor
```

Pseudocode 4.9: Update, adapted for $\mathbf{G_{r-1} = (V, E_{r-1})}$

```
4.9.1   Update(σ_j, G_{r−1}, σ_i)
4.9.2   Input: a cell, σ_j ∈ V, the residual digraph, G_{r−1}, and a cell, σ_i ∈ V
4.9.3   newreach = σ_j.depth
4.9.4   foreach temporarily visited arc (σ_j, σ_k) ∈ E_{r−1}
4.9.5     newreach = min(newreach, σ_k.reach)
4.9.6   endfor
4.9.7   if newreach > σ_j.reach then
4.9.8     σ_j.reach = newreach
4.9.9     foreach temporarily visited arc (σ_k, σ_j) ∈ E_{r−1}
4.9.10      fork Update(σ_k, G_{r−1}, σ_j)
4.9.11    endfor
```

4.9.12 **endif**
4.9.13 **if** σ_i.reach $> \sigma_j$.depth **then**
4.9.14 **fork** Discard(σ_i, G_{r-1})
4.9.15 **endif**

Example 4.4. Figure 4.6 illustrates how the depth and reach-numbers are initially set during forward moves and dynamically adjusted (decreased) during backtrack moves.

(a) In round 1, search path $\tau = \sigma_1.\sigma_2.\sigma_3.\sigma_4.\sigma_5.\sigma_6.\sigma_4$ attempts to visit the already visited cell σ_4 (using Cidon's optimisation, it will not actually visit σ_4). The reach-number of each cell on the search path is still the same as its depth, σ_i.reach $= \sigma_i$.depth $= i, i \in [1, 6]$.

(b) Later, search path $\sigma_1.\sigma_2.\sigma_3.\sigma_4$ backtracks to σ_4. Cells to which the search path has backtracked, σ_6, σ_5 and σ_4, have updated their reach-numbers: σ_6.reach $=$ min$(\sigma_6$.depth$, \sigma_4$.reach$) = 3$; σ_5.reach $=$ min$(\sigma_5$.depth$, \sigma_6$.reach$) = 3$ and σ_4.reach $=$ min$(\sigma_4$.depth$, \sigma_5$.reach$) = 3$.

(c) Search path $\sigma_1.\sigma_2.\sigma_3.\sigma_4.\sigma_7$ moves forward to σ_7, so σ_7.reach $= \sigma_7$.depth $= 4$.

(d) Search path $\sigma_1.\sigma_2.\sigma_3.\sigma_4$ backtracks again to σ_4, so σ_4.reach $=$ min$(\sigma_4$.depth, σ_5.reach$, \sigma_7$.reach$) = 3$ (unchanged). Because σ_7.reach $= 4 > \sigma_4$.depth $= 3$, σ_4 sends a discard notification to σ_7.

(e) Cell σ_7 is discarded, σ_7.reach $= \infty$, and sends an update notification to its st-predecessor, σ_4. Search path $\sigma_1.\sigma_2.\sigma_3$ backtracks to σ_3, so σ_3.reach $=$ min $(\sigma_3$.depth$, \sigma_4$.reach$) = 2$ (unchanged). Because σ_4.reach $= 3 > \sigma_3$.depth $= 2$, σ_3 sends a discard notification to σ_4.

(f) Cell σ_4 is discarded, σ_4.reach $= \infty$, and thus ignores σ_7's update notification; σ_4 further sends a discard notification to its st-successor, σ_5, and an update notification to its st-predecessor, σ_6.

(g) Cell σ_6 updates its reach-number, σ_6.reach $=$ min$(\sigma_6$.depth$, \sigma_4$.reach$) = 5$, and sends an update notification to σ_5. Cell σ_5 is discarded, σ_5.reach $= \infty$, and will ignore σ_6's update notification in the next step; σ_5 further sends a discard notification to its st-successor, σ_6.

(h) Cell σ_6 is discarded, σ_6.reach $= \infty$. Later, search path $\sigma_1.\sigma_2.\sigma_3.\sigma_8$ *succeeds* and confirms back to the source cell. All discard notifications reach their targets before the start of round 2. Subsequent searches will use a trimmed digraph, which will speed up the algorithm. Thus, in round 2, search path, τ', will not needlessly probe the *discarded* cells, σ_5, σ_6, σ_4 and σ_7.

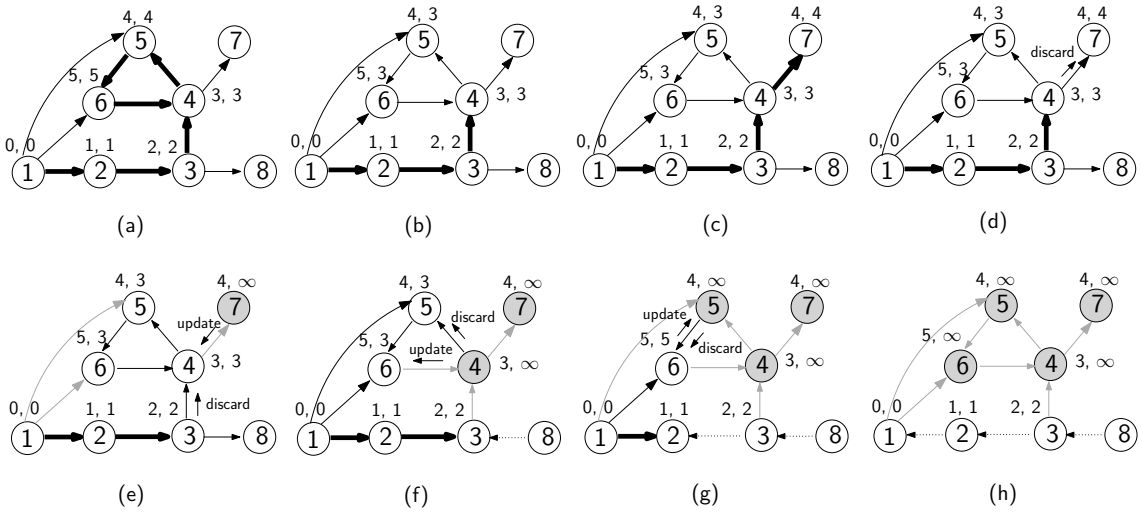


Figure 4.6: How the depth and reach-numbers are initially set during forward moves and dynamically adjusted (decreased) during backtrack moves. Thin arcs: original arcs; thick arcs: search path arcs; gray arcs: discarded arcs; dotted arcs: reversed path arcs; thin arrows beside arcs: discard or update notifications; pair "`depth`, `reach`" beside each cell: a cell's depth and reach-number; gray cells: discarded cells.

In contrast, DFS-Edge-B, which uses a different idea, cannot detect "dead" cells during successful rounds. Its round 1 search path, τ, follows the same route as in DFS-Edge-C, without triggering any discard notification. Therefore, the round 2 search path, τ', will needlessly visit again cells σ_5, σ_6, σ_4 and σ_7.

In this example, DFS-Edge-C shows better performance than DFS-Edge-B.

DFS-Edge-C can detect all "dead" cells identified by DFS-Edge-B. However, because of pruning propagating delays, DFS-Edge-C does not effectively discard them as fast as possible; briefly, it does not show the same runtime performance. Thus, a cell may be visited by the next search process *before* it gets its due discard notification. To solve this problem, a cell immediately backtracks (line 4.7.5) once discarded (by its due discard notification).

Example 4.5. Figure 4.7 shows an example of a cell visited *before* getting its due discard notification.

(a) In round 1, search path $\tau = \sigma_1.\sigma_2.\sigma_4.\sigma_5.\sigma_6.\sigma_7.\sigma_8.\sigma_9.\sigma_{10}$, backtracks to the source cell, σ_1, cells $\sigma_i, i \in \{2\} \cup [4, 10]$ can be discarded, because $\sigma_2.reach = 1 > \sigma_1.\texttt{depth} = 0$. Cell σ_1 triggers a discard notification, which follows the *same path* as the backtracked search τ.

(b) In round 2, the *new* search path τ', $\tau' = \sigma_1.\sigma_3$ visits σ_7 *before* σ_7 receives its due discard notification and then continues further to σ_{10}.

(c) Later, cell σ_7 receives its due discard notification (started in round 1), so it discards itself and sends an overdue *backtrack* token to σ_3, which starts looking for another direction to continue path τ'. However, several steps have been lost exploring "dead" cells (which were not aware of this).

(d) Finally, after this mentioned delay, search path τ', $\tau' = \sigma_1.\sigma_3.\sigma_{11}$, reaches σ_{11} and becomes a new augmenting path.

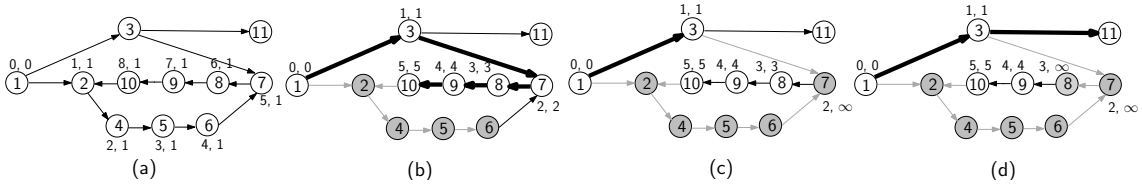


Figure 4.7: An example of a cell visited before receiving its due discard notification. Thin arcs: original arcs; thick arcs: search path arcs; gray arcs: discarded arcs; pair "`depth`, `reach`" beside each cell: a cell's depth and reach-number; gray cells: discarded cells.

In round 1, DFS-Edge-B's search path, τ, follows the same route as in DFS-Edge-C. When τ backtracks to σ_1, DFS-Edge-B initiates an after-failure reset, which is propagated as a *broadcast*, travelling on *shortest paths*, reaching σ_7 on path $\sigma_1.\sigma_3.\sigma_7$. When the immediately following round 2 search path τ' reaches σ_3, $\tau' = \sigma_1.\sigma_3$, it avoids σ_7, which is already discarded. In the next step, the search path τ' reaches σ_{11}, $\tau' = \sigma_1.\sigma_3.\sigma_{11}$, and becomes a new augmenting path, which is faster than in DFS-Edge-C.

In this example, due to its pruning propagating delay, DFS-Edge-C shows worse performance than DFS-Edge-B.

The following result consolidates the above arguments.

Theorem 4.7. Algorithm DFS-Edge-C finds a maximum cardinality set of edge-disjoint paths.

The xP specification of DFS-Edge-C is omitted here; it is contained in its extension, DFS-Edge-D.

DFS-Edge-C* [22] intentionally restricts DFS-Edge-C by *omitting* to discard "dead" cells found in *failed* rounds, i.e. it only discards "dead" cell found in *successful* rounds. This version is used specifically for performance test later in Section 4.7.

4.3.6 Algorithm DFS-Edge-D

Algorithm DFS-Edge-D proposed by ElGindy, Nicolescu and Wu in our joint work [22] combines DFS-Edge-B and C, which discards all "dead" cells that are detected by

B and C. Its Pseudocode 4.10 adds only one line to Pseudocode 4.6. DFS-Edge-D uses two end-of-round resets, as in DFS-Edge-B (§ 4.3.4): (1) an after-success reset, which resets all temporarily visited cells and arcs to unvisited, and (2) an after-failure reset, which sets all temporarily visited cells and arcs as permanently visited (to be discarded).

Pseudocode 4.10: DFS-Edge-D

Same as Pseudocode 4.6, except that one line is added between lines 4.6.14 and 15:

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

DFS-Edge-D is a combination of DFS-Edge-B and C, so the following result is straightforward:

Theorem 4.8. Algorithm DFS-Edge-D finds a maximum cardinality set of edge-disjoint paths.

xP Specification 4.3: DFS-Edge-D

Input: All cells start in the same initial state, S_0, with the same set of rules, without any topological awareness (they do not even know their neighbours). 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 state, S_{60}. 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: disjoint path predecessors, d'_j's, and disjoint path successors, d''_k's.

Symbols and states

Our xP Specification 4.3 of DFS-Edge-D uses the same symbols and states as xP Specification 4.2 of DFS-Edge-B, plus a few more specific to the idea of DFS-Edge-C.

Cell σ_i uses the following symbols to record its states:

- $h(c^l)$ records its depth, l;
- $m(c^l)$ is the intermediate or final computed result of its reach-number;
- v indicates that it is temporarily visited;
- w indicates that it is permanently visited;
- f indicates that it is the frontier cell;

- u indicates that it is receiving an update.

Cell σ_i uses the following symbols to record its relationships with its neighbour cells, σ_j and σ_k:

- $o_j(c^l)$ records σ_j's reach-number, l;
- z''_k records a previous st-successor.

Cell σ_i sends out the following symbols:

- w_i is its permanently visited notification;
- $o'_i(c^l)$ is its reach-number, l;
- u_i is an update notification;
- x is a discard notification.

For cell σ_i, a forward token is indicated by $f_j\ v$, while a backtrack token is indicated by $f_j\ v$. Cell σ_i's st-successors are indicated by (1) $n''_k\ v_k$ (forward mode) and (2) $d'_j\ v_j$ (push-back mode). No st-predecessors are recorded, so update notifications are broadcast and each receiver only accepts the update notification from its st-successor. Thus, by using this additional housekeeping, we properly send discard notifications (rules 4.5.14–4.5.15) and accept update notifications (rules 4.5.4–5).

The matrix R of xP Specification 4.3, $\Pi_1; \Pi_2$, is constructed by sequential composition of Π_1 and Π_2, where Π_1 is SynchNDD (§ 4.2) and Π_2 is the disjoint path search algorithm consisting of eleven vectors, informally presented in five groups, according to their functionality and applicability.

Π_1: SynchNDD (see Section 4.2)

Π_2: Disjoint path search

2. Initial differentiation (S_3)

2.1.

2.1.1. $S_3\ \rightarrow_{\texttt{min.min}}\ S_{10}\ f\ h(\lambda)\ (w_i\ o_i(\lambda))\updownarrow\ \mid \iota_i\ s$

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

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

3. Source cell (S_{10})

3.1.

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

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

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

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

3.1.5. $S_{10}\ f_k\ \rightarrow_{\texttt{max.max}}\ S_{40}$

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

4. Intermediate cells (S_{20})

4.1. Finalisation

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

4.2. Frontier

4.2.1. $S_{20}\ \rightarrow_{\texttt{min.min}}\ S_{20}\ o_i(X)\ (o_i(X))\updownarrow\ \mid \iota_i\ f_j\ h(X)\ \neg\ w\ v$

4.2.2. $S_{20}\ f_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v\ s'_j\ (v_i)\updownarrow\ f \mid \iota_i\ \neg\ w\ v$

4.2.3. $S_{20}\ o_k(X)\ \rightarrow_{\texttt{max.max}}\ S_{20} \mid o_k(X)\ f_k\ v$

4.2.4. $S_{20}\ o'_k(X)\ \rightarrow_{\texttt{max.max}}\ S_{20} \mid o'_k(X)\ f_k\ v$

4.2.5. $S_{20}\ o_k(X)\ o'_k(Y)\ \rightarrow_{\texttt{min.min}}\ S_{20}\ o_k(Y) \mid f_k\ v$

4.2.6. $S_{20}\ f_k\ s''_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ f\ z''_k \mid v$

4.2.7. $S_{20}\ f_k\ \rightarrow_{\texttt{max.max}}\ S_{20}$

4.2.8. $S_{20}\ \rightarrow_{\texttt{min}}\ S_{20}\ m(X) \mid h(X)\ f$

4.2.9. $S_{20}\ m(XY)\ \rightarrow_{\texttt{max.min}}\ S_{20}\ m(X) \mid o_k(X)\ n''_k\ v_k\ f$

4.2.10. $S_{20}\ m(XY)\ \rightarrow_{\texttt{max.min}}\ S_{20}\ m(X) \mid o_j(X)\ d'_j\ v_j\ f$

4.2.11. $S_{20}\ m(XY)\ o_i(X)\ \rightarrow_{\texttt{min.min}}\ S_{20}\ o_i(X) \mid \iota_i\ f$

4.2.12. $S_{20}\ m(X)\ o_i(XY)\ \rightarrow_{\texttt{min.min}}\ S_{20}\ o_i(X)\ (o'_i(X))\updownarrow\ \mid \iota_i\ f$

4.2.13. $S_{20} \rightarrow_{\texttt{min.min}}\ S_{20}(x)\updownarrow_k\ \mid \iota_i\ h(X)\ o_k(XY)\ v\ f\ \neg\ o_k(X)\ w_k$

4.2.14. $S_{20}\ \rightarrow_{\texttt{min}}\ S_{20}\ v \mid f\ \neg\ v$

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

4.2.16. $S_{20}\ f\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v_j\ s''_j\ (f_i\ h(Xc))\uparrow_j\ \mid \iota_i\ d'_j\ h(X)\ \neg\ w_j\ v_j$

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

4.3. Path confirmation

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

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

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

4.4. End-of-round resets

4.4.1. $S_{20} \rightarrow_{\texttt{min}} S_{40}\ (q)\updownarrow \mid q$

4.4.2. $S_{20} \rightarrow_{\texttt{min}} S_{40}\ w \mid q\ v \neg w$

4.4.3. $S_{20} \rightarrow_{\texttt{max.min}} S_{40}\ w_k \mid q\ v_k \neg w_k$

4.4.4. $S_{20} \rightarrow_{\texttt{min}} S_{40}\ (r)\updownarrow \mid r$

4.5. Update and Discard

//Update

4.5.1. $S_{20}\ o_j(X) \rightarrow_{\texttt{max.max}} S_{20} \mid o_j(X) \neg w$

4.5.2. $S_{20}\ o'_j(X) \rightarrow_{\texttt{max.max}} S_{20} \mid o'_j(X) \neg w$

4.5.3. $S_{20}\ o_j(X)\ o'_j(Y) \rightarrow_{\texttt{min.min}} S_{20}\ o_j(Y) \neg w$

4.5.4. $S_{20} \rightarrow_{\texttt{min.min}} S_{20}\ u \mid u_k\ n''_k\ v_k \neg w$

4.5.5. $S_{20} \rightarrow_{\texttt{min.min}} S_{20}\ u \mid u_j\ d'_j\ v_j \neg w$

4.5.6. $S_{20} \rightarrow_{\texttt{min}} S_{20}\ m(X) \mid h(X)\ u \neg w$

4.5.7. $S_{20}\ m(XY) \rightarrow_{\texttt{max.min}} S_{20}\ m(X) \mid o_k(X)\ n''_k\ v_k\ u \neg w$

4.5.8. $S_{20}\ m(XY) \rightarrow_{\texttt{max.min}} S_{20}\ m(X) \mid o_j(X)\ d'_j\ v_j\ u \neg w$

4.5.9. $S_{20}\ m(X)\ o_i(XY) \rightarrow_{\texttt{min.min}} S_{20}\ o_i(XY) \mid u \neg w$

4.5.10. $S_{20}\ m(XY)\ o_i(X) \rightarrow_{\texttt{min.min}} S_{20}\ o_i(XY)\ (o'_i(XY)\ u_i)\updownarrow \mid \iota_i\ u \neg w$

4.5.11. $S_{20} \rightarrow_{\texttt{min.min}} S_{20}\ (x)\updownarrow_j \mid \iota_i\ u\ h(X)\ o_j(XY) \neg o_k(X)\ w_j\ w$

4.5.12. $S_{20}\ u_j \rightarrow_{\texttt{max.max}} S_{20}$

4.5.13. $S_{20}\ o'_j(X) \rightarrow_{\texttt{max.max}} S_{20}$

//Discard

4.5.14. $S_{20} \rightarrow_{\texttt{max.min}} S_{20}\ (x)\updownarrow_k \mid \iota_i\ n''_k\ v_k\ x \neg w_k\ w$

4.5.15. $S_{20} \rightarrow_{\texttt{max.min}} S_{20}\ (x)\updownarrow_j \mid \iota_i\ d'_j\ v_j\ x \neg w_j\ w$

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

4.5.17. $S_{20}\ x \rightarrow_{\texttt{min.min}} S_{20}\ w\ o_i(\infty)\ (w_i\ o'_i(\infty)\ u_i)\updownarrow \mid \iota_i \neg w$

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

4.5.19. $S_{20}\ s'_j\ s''_k \rightarrow_{\texttt{min.min}} S_{20}\ (f_i)\updownarrow_j \mid \iota_i\ w$

4.5.20. $S_{20}\ f_j \rightarrow_{\texttt{min.min}} S_{20}\ (f_i)\updownarrow_j \mid \iota_i\ w$

4.5.21. $S_{20}\ u \rightarrow_{\texttt{max}} S_{20}$

5. Target cell (S_{30})

5.1.

5.1.1. $S_{30}\ g \rightarrow_{\texttt{min}} S_{50}\ (g)\updownarrow$

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

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

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

6. All cells (S_{40}, S_{41}, S_{50})

6.1. Transit to the end of a search round

6.1.1. $S_{40} \rightarrow_{\texttt{min}} S_{41}$

6.2. End of each search round

6.2.1. $S_{41}\ v_k \rightarrow_{\texttt{max.max}} S_3$

6.2.2. $S_{41}\ v \rightarrow_{\texttt{max}} S_3$

6.2.3. $S_{41}\ u_j \rightarrow_{\texttt{max.max}} S_3$

6.2.4. $S_{41}\ c_k \rightarrow_{\texttt{max.max}} S_3$

6.2.5. $S_{41}\ o_j(X) \rightarrow_{\texttt{max.max}} S_3$

6.2.6. $S_{41}\ o'_j(X) \rightarrow_{\texttt{max.max}} S_3$

6.2.7. $S_{41}\ a \rightarrow_{\texttt{max}} S_3$

6.2.8. $S_{41}\ q \rightarrow_{\texttt{max}} S_3$

6.2.9. $S_{41}\ r \rightarrow_{\texttt{max}} S_3$

6.2.10. $S_{41} \rightarrow_{\texttt{min}} S_3$

6.3. End of the algorithm

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

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

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

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

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

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

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

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

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

Initial and final configurations

Table 4.7 shows the initial and final configurations of xP Specification 4.3 for Figure 4.6.

Rule explanations

DFS-Edge-D combines DFS-Edge-B and C, so we only explain the rules for implementing the optimisation rules of DFS-Edge-C discussed in Section 4.3.5.

Table 4.7: Initial and final configurations of xP Specification 4.3 for Figure 4.6.

Cell	σ_1	σ_2	σ_3	σ_4
Initial	$S_0\ \iota_1\ s$	$S_0\ \iota_2$	$S_0\ \iota_3$	$S_0\ \iota_4$
Final	$S_{60}\ \iota_1\ s\ d''_2$	$S_{60}\ \iota_2\ d'_1\ d''_3$	$S_{60}\ \iota_3\ d'_2\ d''_8$	$S_{60}\ \iota_4$
Cell	σ_5	σ_6	σ_7	σ_8
Initial	$S_0\ \iota_5$	$S_0\ \iota_6$	$S_0\ \iota_7$	$S_0\ \iota_8\ t$
Final	$S_{60}\ \iota_5$	$S_{60}\ \iota_6$	$S_{60}\ \iota_7$	$S_{60}\ \iota_8\ t\ d'_3$

At the start of each search round, the source cell, σ_s, sets its depth as $h(\lambda)$ and broadcasts its permanently visited notification, w_s, and its reach-number, $o_s(\lambda)$, to all its neighbours, becoming a frontier cell, f (rule 2.1.1).

If sending a *forward* token, a current frontier cell of depth $h(X)$ also sends an incremented depth, $h(Xc)$ together with the token (rule 3.1.1 for the source cell, rules 4.2.15–16 for an intermediate cell).

1. When an unvisited cell, σ_i, receives a forward token, f_j, and a depth, $h(X)$,

   (a) σ_i marks itself as temporarily visited by v and broadcasts v_i to all its neighbours (rule 4.2.1);

   (b) σ_i sets its depth as $h(X)$, its reach-number as $o_i(X)$ and broadcasts $o_i(X)$ to all its neighbours (rule 4.2.2).

2. When cell σ_i receives a backtrack token, f_k, and σ_k's reach-number, $o'_k(X)$ or when cell σ_i avoids revisiting a temporarily visited child, σ_k (in both cases, σ_i contains f),

   (a) σ_i updates its recorded σ_k's reach-number (rules 4.2.3–5) and computes its reach-number as the minimum of its depth, $h(X)$, and the reach-numbers of its st-successors, $n''_k v_k$ and $d'_k v_k$ (rules 4.2.8–10); if σ_i's reach-number is decreased, it broadcasts its reach-number (rules 4.2.11–12);

   (b) if σ_i's depth is less than σ_k's reach-number, it sends a discard notification, x, to σ_k (rule 4.2.13).

3. When cell σ_i receives a discard notification, x,

   (a) σ_i marks itself as permanently visited by w and broadcasts w_i to all its neighbours (rule 4.5.17);

   (b) σ_i sets its reach-number as $o_i(\infty)$ (rule 4.5.17);

   (c) σ_i broadcasts u_i and $o'_i(\infty)$ to all its neighbours (rule 4.5.17);

   (d) σ_i sends discard notifications, x, to all its st-successors (rules 4.5.14–16).

Specifically, if cell σ_i that is discarded by an overdue discard notification is on the current search path, indicated by f_j or s'_j, it immediately sends a backtrack token, f_i, to its sp-predecessor, σ_j, offering some speed up (rules 4.5.19–20).

4. When cell σ_i receives an update notification, u_k, and $o'_k(Y)$ from its st-successor, σ_k (rules 4.5.1–5), it updates σ_k's reach-number as $o_k(Y)$ and computes its reach-number as the minimum of its depth, $h(X)$, and the reach-numbers of its st-successors, $n''_k v_k$ and $d''_k v_k$ (rules 4.5.6–8). If σ_i's reach-number is increased (rule 4.5.9),

(a) σ_i broadcasts an update notification, u_i, and its new reach-number, $o'_i(Z)$ (rule 4.5.10);

(b) if σ_i's depth is less than σ_j's reach-number, σ_i sends a discard notification, x, to σ_j (rule 4.5.11).

Partial traces

Table 4.8 shows the partial traces of xP Specification 4.3 for cell σ_4 in Figure 4.6. Omitted symbols $(\dots)$ are $\iota_4\ n'_3\ n'_6\ n''_5\ n''_7$. As previously mentioned, in the actual algorithm, a cell simply *broadcasts* its update notification and reach-number instead of sending them to the only concerned cells as in Figure 4.6. Therefore, row (h) show that σ_4 receives update notifications from σ_5 and σ_6 but ignores them because it is already discarded.

Table 4.8: Partial traces of xP Specification 4.3 for cell σ_4 in Figure 4.6.

Fig.	Evolution	Content
(a)		$v\ v_3\ v_5\ v_6\ s'_3\ s''_5$ $h(c^3)\ o_3(c^2)\ o_4(c^3)\ o_5(c^4)\ o_6(c^5)$
(b)	$\{f_5\ o'_5(c^3)\}\ s''_5\ o_5(c^4) \Rightarrow$ $z''_5\ v_7\ s''_7\ \{f_4\}_7$	$v\ v_3\ v_5\ v_6\ v_7\ s'_3\ s''_7\ z''_5$ $h(c^3)\ o_3(c^2)\ o_4(c^3)\ o_5(c^3)\ o_6(c^3)$
(c)		$v\ v_3\ v_5\ v_6\ v_7\ s'_3\ s''_7\ z''_5$ $h(c^3)\ o_3(c^2)\ o_4(c^3)\ o_5(c^3)\ o_6(c^3)$
(d)	$\{f_7\ o_7(c^4)\}\ s'_3\ s''_7 \Rightarrow$ $z''_7\ \{f_4\}_3\ \{x\}_7$	$v\ v_3\ v_5\ v_6\ v_7\ z''_5\ z''_7$ $h(c^3)\ o_3(c^2)\ o_4(c^3)\ o_5(c^3)\ o_6(c^3)\ o_7(c^4)$
(e)		$v\ v_3\ v_5\ v_6\ v_7\ z''_5\ z''_7$ $h(c^3)\ o_3(c^2)\ o_4(c^3)\ o_5(c^3)\ o_6(c^3)\ o_7(c^4)$
(f)	$\{x\ u_7\ w_7\ o'_7(\infty)\}\ z''_5\ z''_7\ o_4(c^3) \Rightarrow$ $w\ w_7\ o_4(\infty)\ o_7(\infty)\ \{x\}_5\ \{u_4\ w_4\ o_4(\infty)\}_{3,5,6,7}$	$v\ v_3\ v_5\ v_6\ v_7\ w\ w_7$ $h(c^3)\ o_3(c^2)\ o_4(\infty)\ o_5(c^3)\ o_6(c^3)\ o_7(\infty)$
(g)		$v\ v_3\ v_5\ v_6\ v_7\ w\ w_7$ $h(c^3)\ o_3(c^2)\ o_4(\infty)\ o_5(c^3)\ o_6(c^3)\ o_7(\infty)$
(h)	$\{u_5\ w_5\ o'_5(\infty)\ u_6\ o'_6(c^5)\} \Rightarrow w_5$	$v\ v_3\ v_5\ v_6\ v_7\ w\ w_5\ w_7$ $h(c^3)\ o_3(c^2)\ o_4(\infty)\ o_5(c^3)\ o_6(c^3)\ o_7(\infty)$

4.4 DFS-based Node-disjoint Paths Algorithm

Dinneen et al. [18] has proposed DFS-Node-A*, a node-disjoint version of DFS-Edge-A*, which is the first node-disjoint paths solution in P systems. It simulates node-splitting (§ 4.1.2) by (1) constraining in- and out-flow capacities of each intermediate cell to one and (2) having entry and exit visited marks for each intermediate cell.

Following our joint work [52], this section presents DFS-Node-B, which is a node-disjoint version of DFS-Edge-B. It is a faster node-disjoint paths solution and also uses the node-splitting simulation.

4.4.1 Algorithm DFS-Node-B

Algorithm DFS-Node-B is a node-disjoint version of DFS-Edge-B (§ 4.3.4), which solves the node-disjoint problem using node-splitting simulation (§ 4.1.2); the following result is straightforward:

Theorem 4.9. Algorithm DFS-Node-B finds a maximum cardinality set of node-disjoint paths.

xP Specification 4.4: DFS-Node-B

Input: As in the edge-disjoint paths algorithm of xP Specification 4.2.

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

Symbols and states

Our xP Specification 4.4 of DFS-Node-B uses the same symbols and states as xP Specification 4.2 of DFS-Edge-B, plus a few more specific.

Cell σ_i uses the following symbols to record its relationship with its neighbour cell, σ_j:

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

Cell σ_i uses the following symbols to indicate its states:

- $v \to v'$ and v'' indicate that its entry and exit nodes are temporarily visited, respectively;

- $w \to w'$ and w'' indicate that its entry and exit nodes are permanently visited, respectively;
- c indicates that σ_i must next constrain its in-flow to one.

The matrix R of xP Specification 4.2, $\Pi_1; \Pi_2$, is constructed by sequential composition of Π_1 and Π_2, where Π_1 is SynchNDD (§ 4.2) and Π_2 is the disjoint path search algorithm consisting of ten vectors, informally presented in five groups, according to their functionality and applicability.

Π_1: SynchNDD (see Section 4.2)

Π_2: Disjoint path search

2. Initial differentiation (S_3)

2.1.

2.1.1. $S_3 \ \to_{\texttt{min}} \ S_{10}\ f \mid s$

2.1.2. $S_3 \ \to_{\texttt{min}} \ S_{30} \mid t$

2.1.3. $S_3 \ \to_{\texttt{min}} \ S_{20}$

3. Source cell (S_{10})

3.1.

3.1.1. $S_{10}\ f \ \to_{\texttt{min.min}} \ S_{10}\ s''_k\ (v'_i\ v''_i)\updownarrow \ (f_i)\downarrow_k \ \mid n''_k \neg w_k\ v_k$

3.1.2. $S_{10}\ a\ s''_k\ n''_k \ \to_{\texttt{min.min}} \ S_{40}\ r\ d''_k\ (r)\updownarrow$

3.1.3. $S_{10}\ a \ \to_{\texttt{max}} \ S_{40}$

3.1.4. $S_{10}\ f_k\ s''_k\ n''_k \ \to_{\texttt{min.min}} \ S_{40}\ q\ (q)\updownarrow$

3.1.5. $S_{10}\ v'_k \ \to_{\texttt{max.min}} \ S_{40}\ w'_k \mid q \neg w'_k$

3.1.6. $S_{10}\ v''_k \ \to_{\texttt{max.min}} \ S_{40}\ w''_k \mid q \neg w''_k$

3.1.7. $S_{10}\ f_k \ \to_{\texttt{max.max}} \ S_{40}$

3.1.8. $S_{10}\ f \ \to_{\texttt{min}} \ S_{50}\ (g)\updownarrow$

4. Intermediate cells (S_{20})

4.1. Finalisation

4.1.1. $S_{20}\ g \ \to_{\texttt{min}} \ S_{50}\ (g)\updownarrow$

4.2. Frontier

4.2.1. $S_{20}\ f_k \ \to_{\texttt{min.min}} \ S_{20}\ v''\ s'_k\ (v''_i)\updownarrow \ f \mid \iota_i\ d''_k \neg w''\ v''$

4.2.2. $S_{20}\ f_j \ \to_{\texttt{min.min}} \ S_{20}\ v'\ c\ s'_j\ (v'_i)\updownarrow \ f \mid \iota_i\ d \neg w'\ v'\ w''\ v''\ d''_j$

4.2.3. $S_{20}\ f_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v'\ s'_j\ (v'_i)\updownarrow\ f \mid \iota_i \neg d\ w'\ v'\ w''\ v''\ d''_j$

4.2.4. $S_{20}\ f_k\ s''_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ c\ f \mid d'_k$

4.2.5. $S_{20}\ f_k\ s''_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ f$

4.2.6. $S_{20}\ f_k\ \rightarrow_{\texttt{max.max}}\ S_{20}$

4.2.7. $S_{20}\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v'' \mid f\ n''_k \neg c\ w'_k\ v'_k\ w''_k\ v''_k\ d'_k\ d''_k\ v''$

4.2.8. $S_{20}\ f\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v'_k\ s''_k\ (v''_i)\updownarrow\ (f_i)\downarrow_k \mid \iota_i\ n''_k \neg c\ w'_k\ v'_k\ w''_k\ v''_k\ d'_k\ d''_k$

4.2.9. $S_{20}\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v' \mid f\ d'_k \neg w''_k\ v''_k\ v'$

4.2.10. $S_{20}\ f\ \rightarrow_{\texttt{min.min}}\ S_{20}\ v''_k\ s''_k\ (v'_i)\updownarrow\ (f_i)\uparrow_k \mid \iota_i\ d'_k \neg w''_k\ v''_k$

4.2.11. $S_{20}\ f\ s'_j\ \rightarrow_{\texttt{min.min}}\ S_{20}\ (f_i)\downarrow_j \mid \iota_i\ d''_j$

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

4.2.13. $S_{20}\ c\ \rightarrow_{\texttt{min}}\ S_{20}$

4.3. Path confirmation

4.3.1. $S_{20}\ a\ \rightarrow_{\texttt{max}}\ S_{20} \mid a$

4.3.2. $S_{20}\ a\ s'_j\ s''_k\ \rightarrow_{\texttt{min.min}}\ S_{20}\ d\ d'_j\ d''_k\ (a)\downarrow_j \mid d''_j \neg d'_k$

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

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

4.4. End-of-round resets

4.4.1. $S_{20}\ \rightarrow_{\texttt{min}}\ S_{40}\ (q)\updownarrow \mid q$

4.4.2. $S_{20}\ v'\ \rightarrow_{\texttt{min}}\ S_{40}\ w' \mid \iota_i\ q \neg w'$

4.4.3. $S_{20}\ v''\ \rightarrow_{\texttt{min}}\ S_{40}\ w'' \mid \iota_i\ q \neg w''$

4.4.4. $S_{20}\ v'_k\ \rightarrow_{\texttt{max.min}}\ S_{40}\ w'_k \mid q \neg w'_k$

4.4.5. $S_{20}\ v''_k\ \rightarrow_{\texttt{max.min}}\ S_{40}\ w''_k \mid q \neg w''_k$

4.4.6. $S_{20}\ \rightarrow_{\texttt{min}}\ S_{40}\ (r)\updownarrow \mid r$

5. Target cell (S_{30})

5.1.

5.1.1. $S_{30}\ g\ \rightarrow_{\texttt{min}}\ S_{50}\ (g)\updownarrow$

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

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

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

6. All cells (S_{40}, S_{41}, S_{50})

6.1.

6.1.1. $S_{40} \rightarrow_{\texttt{min}} S_{41}$

6.2. End of each search round

6.2.1. $S_{41}\ v'_k \rightarrow_{\texttt{max.max}} S_3$

6.2.2. $S_{41}\ v''_k \rightarrow_{\texttt{max.max}} S_3$

6.2.3. $S_{41}\ v' \rightarrow_{\texttt{max}} S_3$

6.2.4. $S_{41}\ v'' \rightarrow_{\texttt{max}} S_3$

6.2.5. $S_{41}\ a \rightarrow_{\texttt{max}} S_3$

6.2.6. $S_{41}\ q \rightarrow_{\texttt{max}} S_3$

6.2.7. $S_{41}\ r \rightarrow_{\texttt{max}} S_3$

6.2.8. $S_{41} \rightarrow_{\texttt{min}} S_3$

6.3. End of the algorithm

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

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

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

6.3.4. $S_{50}\ w'_k \rightarrow_{\texttt{max.max}} S_{50}$

6.3.5. $S_{50}\ w''_k \rightarrow_{\texttt{max.max}} S_{50}$

6.3.6. $S_{50}\ v'_k \rightarrow_{\texttt{max.max}} S_{50}$

6.3.7. $S_{50}\ v''_k \rightarrow_{\texttt{max.max}} S_{50}$

6.3.8. $S_{50}\ w' \rightarrow_{\texttt{max}} S_{50}$

6.3.9. $S_{50}\ w'' \rightarrow_{\texttt{max}} S_{50}$

6.3.10. $S_{50}\ v' \rightarrow_{\texttt{max}} S_{50}$

6.3.11. $S_{50}\ v'' \rightarrow_{\texttt{max}} S_{50}$

6.3.12. $S_{50}\ d \rightarrow_{\texttt{max}} S_{50}$

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

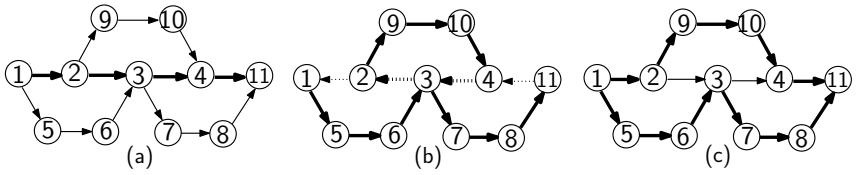


Figure 4.8: An example of node-disjoint paths.

Table 4.9: Initial and final configurations of xP Specification 4.4, for Figure 4.8.

Cell	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
Initial	$S_0\ \iota_1\ s$	$S_0\ \iota_2$	$S_0\ \iota_3$	$S_0\ \iota_4$	$S_0\ \iota_5$	$S_0\ \iota_6$
Final	$S_{60}\ \iota_1\ s\ d''_2\ d''_5$	$S_{60}\ \iota_2\ d'_1\ d''_9$	$S_{60}\ \iota_3\ d'_6\ d''_7$	$S_{60}\ \iota_4\ d'_{10}\ d''_{11}$	$S_{60}\ \iota_5\ d'_1\ d''_6$	$S_{60}\ \iota_6\ d'_5\ d''_3$

Cell	σ_7	σ_8	σ_9	σ_{10}	σ_{11}
Initial	$S_0\ \iota_7$	$S_0\ \iota_8$	$S_0\ \iota_9$	$S_0\ \iota_{10}$	$S_0\ \iota_{11}\ t$
Final	$S_{60}\ \iota_7\ d'_3\ d''_8$	$S_{60}\ \iota_8\ d'_7\ d''_{11}$	$S_{60}\ \iota_9\ d'_2\ d''_{10}$	$S_{60}\ \iota_{10}\ d'_9\ d''_4$	$S_{60}\ \iota_{11}\ t\ d'_4\ d'_8$

Initial and final configurations

Table 4.9 shows the initial and final configurations of xP Specification 4.4 for Figure 4.8.

Rule explanations

As discussed before, DFS-Node-B is a node-disjoint version of DFS-Edge-B, so we only explain the rules of the node-splitting simulation (§ 4.1.2). An intermediate cell, σ_i, handles its received token using the following rules corresponding to the token handling rules in Section 4.3, which are adapted for the node-splitting simulation.

(I) Rules 4.2.1–3.

Forward mode: when cell σ_i, which is unvisited on both its entry and exit nodes, $\neg\ w''\ v''\ w'\ v'$ (visiting restriction in Section 4.1.2), receives a forward token, f_j, cell σ_i (i) records its sp-predecessor as s'_j, (ii) marks its entry node as visited by v' and (iii) broadcasts v'_i to all its neighbours, becoming a frontier by generating one f (rule 4.2.3). Specifically, if σ_i is on a disjoint path, d, it generates one c, indicating that it must next enforce the in-flow to one (rule 4.2.2).

Push-back mode: when cell σ_i, which is unvisited on its exit node, $\neg\ w''\ v''$, receives a forward token, f_k, from its dp-successor, d''_k, cell σ_i (i) records its sp-predecessor as s'_k, (ii) marks its exit node as visited by v'' and (iii) broadcasts v''_i to all its neighbours, becoming a frontier by generating one f (rule 4.2.1).

(II) Rules 4.2.4–5: when σ_i receives a backtrack token, f_k, from the sp-successor of its entry node, d'_k, it erases s''_k, becomes a frontier by generating one f and generates one c, indicating that it must next enforce the in-flow to one (rule 4.2.4). Otherwise, when cell σ_i receives f_k from its sp-successor that is not the sp-successor of its entry node, s''_k, it erases s''_k and generates one f (rule 4.2.5).

As a frontier, σ_i sends its forward token (in the forward or push-back mode) or backtrack token as follows.

(a') Rules 4.2.7–8: if σ_i has no constraint, $\neg\ c$, and has any child, n''_k, which is (1) unvisited on its entry node, $\neg\ w'_k\ v'_k$, (2) unvisited on its exit node, $\neg\ w''_k\ v''_k$ (visiting restriction in Section 4.1.2) and (3) neither its dp-predecessor nor its dp-successor, $\neg\ d'_k\ d''_k$, cell σ_i (i) sends a forward token, f_i, in the forward mode to σ_k, (ii) records its sp-successor as s''_k, (iii) marks its exit node as visited by v'' if it has not yet been marked as visited, $\neg\ v''$ (rule 4.2.7), (iv) marks σ_k's entry node as visited by v'_k and (v) broadcasts v''_i to all its neighbours.

(a") Rules 4.2.9–10: if σ_i has a constraint, c, and has a dp-predecessor, d'_k, which is unvisited on its exit node, $\neg\ w''_k\ v''_k$, cell σ_i (i) sends a forward token, f_i, in the push-back mode to σ_k, (ii) records its sp-successor as s''_k, (iii) marks its entry node as visited by v' if it is not marked as visited, $\neg\ v'$ (rule 4.2.9), (iv) marks σ_k's exit node as visited by v''_k and (v) broadcasts v'_i to all its neighbours.

(b) Rules 4.2.11–12: otherwise, σ_i sends a backtrack token, f_i, to its sp-predecessor, s'_j.

After sending the (forward or backtrack) token, cell σ_i erases c, if any (rule 4.2.13).

A cell on a disjoint path can be visited (1) only on its entry or exit node or (2) first on its entry node and later on its exit node (visiting restriction in Section 4.1.2). Thus, a visited cell that is on a disjoint path can have one or two pairs of sp-predecessor and sp-successor.

Consider a cell on a disjoint path, which receives a path confirmation, a.

- According to rule 4.3.2, if cell σ_i has a sp-predecessor of its exit node, $s'_j\ d''_j$, it forwards a to σ_j and transforms s'_j into d'_j and the sp-successor pointer of its exit node, s''_k (rather than the sp-successor pointer of its entry node, $\neg\ d'_k$) into d''_k. After pointer transformations, there is only one pair of sp-predecessor and sp-successor (of its entry node) left.

- If the conditions of rule 4.3.2 are not met (rules are applied in the weak priority order), then rule 4.3.3 is considered: if σ_i does not have a sp-predecessor of its exit node, it forwards a to the only one sp-predecessor, σ_j, and transforms s'_j into d'_j and s''_k into d''_k.

A cell that is not on a disjoint path has only one pair of sp-predecessor and sp-successor and thus forwards a path confirmation, also using rule 4.3.3.

Example 4.6. In Figure 4.8(b), search path $\sigma_1.\sigma_5.\sigma_6.\sigma_3.\sigma_2.\sigma_9.\sigma_{10}.\sigma_4.\sigma_3.\sigma_7.\sigma_8.\sigma_{11}$, has visited cell σ_3 twice—once by a forward search on its entry node (rule 4.2.2) and again by a push-back on its exit node (rule 4.2.1). Cell σ_3 contains $\{d'_2, d''_4, s'_6, s''_2, s'_4, s''_7\}$.

(a) On receiving a from σ_7, because σ_3 has a sp-predecessor of its exit node, $s'_4\ d''_4$, it forwards a to σ_4 (rule 4.3.2). Also, σ_3 transforms s'_4 into d'_4 and s''_7 (rather

than the sp-successor pointer of its entry node, s''_2) into d''_7, and next deletes d'_4 and d''_4, as one end of the cancelling arc (rule 4.3.4). At this time, σ_3 contains $\{d'_2, d''_7, s'_6, s''_2\}$.

(b) Later, on receiving a from σ_2, σ_3 forwards a to its only one sp-predecessor, σ_6 (rule 4.3.3). Also, σ_3 transforms s'_6 into d'_6 and s''_2 into d''_2, and next deletes d'_2 and d''_2 as one end of the cancelling arc (rule 4.3.4). At this time, σ_3 contains $\{d'_6, d''_7\}$.

Partial traces

Table 4.10 shows cells' configurations of xP Specification 4.4 for the three snapshots in Figure 4.8. Omitted symbols (...) are cell ID, σ_i, parent pointers, n'_j's, and children pointers, n''_k's, which are the same as in the "Top." row.

Table 4.10: Partial traces of xP Specification 4.4 for Figure 4.8.

Fig.	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
Top.	$S_{10}\ \iota_1 s n''_2 n''_5$	$S_{20}\ \iota_2 n'_1 n''_3 n''_9$	$S_{20}\ \iota_3 n'_2 n'_6 n''_4 n''_7$	$S_{20}\ \iota_4 n'_3 n'_{10} n''_{11}$	$S_{20}\ \iota_5 n'_1 n''_6$	$S_{20}\ \iota_6 n'_5 n''_3$
(a)	$S_{10}\ s d''_2 \dots$	$S_{20}\ d'_1 d''_3 \dots$	$S_{20}\ d'_2 d''_4 \dots$	$S_{20}\ d'_3 d''_{11} \dots$	$S_{20} \dots$	$S_{20} \dots$
(b)	$S_{10}\ s d''_2 s''_5 \dots$	$S_{20}\ d'_1 d''_3$ $s'_3 s''_9 \dots$	$S_{20}\ d'_2 d''_4$ $s'_4 s'_6 s''_2 s''_7 \dots$	$S_{20}\ d'_3 d''_{11}$ $s'_{10} s''_3 \dots$	$S_{20}\ s'_1 s''_6 \dots$	$S_{20}\ s'_5 s''_3 \dots$
(c)	$S_{60}\ s d''_2 d''_5 \dots$	$S_{60}\ d'_1 d''_9 \dots$	$S_{60}\ d'_6 d''_7 \dots$	$S_{60}\ d'_{10} d''_{11} \dots$	$S_{60}\ d'_1 d''_6 \dots$	$S_{60}\ d'_5 d''_3 \dots$

Fig.	σ_7	σ_8	σ_9	σ_{10}	σ_{11}
Top.	$S_{20}\ \iota_7 n'_3 n''_8$	$S_{20}\ \iota_8 n'_7 n''_{11}$	$S_{20}\ \iota_9 n'_2 n''_{10}$	$S_{20}\ \iota_{10} n'_9 n''_4$	$S_{30}\ \iota_{11} t n'_4 n'_8$
(a)	$S_{20} \dots$	$S_{20} \dots$	$S_{20} \dots$	$S_{20} \dots$	$S_{30}\ t d'_4 \dots$
(b)	$S_{20}\ s'_3 s''_8 \dots$	$S_{20}\ s'_7 s''_{11} \dots$	$S_{20}\ s'_2 s''_{10} \dots$	$S_{20}\ s'_9 s''_4 \dots$	$S_{30}\ t d'_4 d'_8 \dots$
(c)	$S_{60}\ d'_3 d''_8 \dots$	$S_{60}\ d'_7 d''_{11} \dots$	$S_{60}\ d'_2 d''_{10} \dots$	$S_{60}\ d'_9 d''_4 \dots$	$S_{60}\ t d'_4 d'_8 \dots$

4.5 BFS-based Edge-disjoint Paths Algorithms

This section presents two BFS-based edge-disjoint paths algorithms:

- BFS-Edge-A proposed by Nicolescu and Wu in our joint work [51, 52] is a distributed version of the classical edge-disjoint paths algorithm, which is based on Edmonds-Karp's maximum flow algorithm [20], and improves its runtime by using a *branch cut* technique.
- BFS-Edge-B proposed by Wu as my own work [65] improves BFS-Edge-A by discarding "dead" cells detected during the *first* search round, which can remain *permanently visited*.

We first describe these two algorithms using high-level *structural parallel* pseudocodes and then present their xP specifications.

Besides the challenges discussed in Section 4.3, BFS-based xP specifications have additional challenging tasks:

- how to detect the search advancing progress;
- how to manage concurrent race conditions, which is similar to the Echo algorithm (see Section 3.2).

4.5.1 BFS Token Handling Rules in Residual Digraphs

BFS-Edge-A and B use BFS to reach the target in a *digraph* by *visit tokens*. An *intermediate* cell handles received visit tokens based on the following *token handling rules*, which are similar to the token handling rules in Section 3.4.1, except that these rules are adapted to work on a virtual residual digraph.

(I) When an *unvisited* cell, σ_i, receives visit tokens (in the forward mode) from its parents or (in the push-back mode) from its dp-successors, cell σ_i (i) selects *one* of the sending cells, σ_j, as its sp-predecessor, (ii) marks itself as visited, becoming a *frontier* cell. Then:

(a') it sends visit tokens (in the forward mode) to all its children, which are neither its dp-predecessors nor its dp-successors, if any; and

(a") it sends visit tokens (in the push-back mode) to all its dp-predecessors, if any.

(II) When *visited* cell, σ_i, receives visit tokens, it ignores the tokens.

4.5.2 Algorithm BFS-Edge-A

Algorithm BFS-Edge-A proposed by Nicolescu and Wu in our joint work [51, 52] is a distributed version of BFS-based edge-disjoint paths algorithm, which is based on Edmonds-Karp's maximum flow algorithm [20] and concurrently searches as many paths as possible. We improve its runtime by using a *branch cut* technique [51, 52], which is critical to make a quick decision when several search paths arrive at the target cell. A search path takes the first *intermediate* cell after the source cell as its own *branch ID*: when several search paths with the *same branch ID* reach the target, only one of them succeeds and the others fail. This technique will be discussed in more detail later.

BFS-Edge-A is described by structural parallel Pseudocodes 4.11 and 4.12 and uses the same global variables as DFS-Edge-A*, plus a few more specific. Global

variable *Paths* (lines 4.11.8–11) represents the set of augmented paths found by a search round. The target cell, σ_t, keeps a local cell variable, $BranchIDs$, (lines 4.11.5 and 4.12.6), which is a growing collection of recorded branch IDs used for filtering incompatible search paths reaching the target in the same round (line 4.12.5). As a consequence of the branch cut technique, *at most one* of the parallel tasks, launched by `parallel foreach` operator of line 4.12.12, succeeds in returning a path. Most of these discussions also apply to the node-disjoint version, BFS-Node-A (§ 4.6.1), which is based on BFS-Edge-A.

Pseudocode 4.11: BFS-Edge-A

```
4.11.1  Input : a digraph G = (V, E); a source cell, σ_s ∈ V;
4.11.2          a target cell, σ_t ∈ V
4.11.3  P_0 = ∅, G_0 = G
4.11.4  r = 0
4.11.5  σ_t.BranchIDs = ∅
4.11.6  while true
4.11.7     set σ_s as visited
4.11.8     Paths = ∅
4.11.9     parallel foreach (σ_s, σ_q) ∈ E_{r-1}
4.11.10       Paths.Add(BFS(q, σ_q, σ_t, G_{r-1}))      // Pseudocode 4.12
4.11.11    endfor
4.11.12    if Paths = ∅ then break
4.11.13    foreach β ∈ Paths
4.11.14       α = σ_s.β
4.11.15       P_{r-1}‾ = (P_{r-1}‾ \ ᾱ^{-1}) ∪ (ᾱ \ P_{r-1}‾^{-1})
4.11.16       G_{r-1} = (V, E_{r-1}), where E_{r-1} = (E \ P_{r-1}‾) ∪ P_{r-1}‾^{-1}
4.11.17    endfor
4.11.18    reset all visited cells to unvisited
4.11.19    r = r + 1
4.11.20 endwhile
4.11.21 Output : P_r, which is a maximum cardinality set of edge-disjoint paths
```

Pseudocode 4.12: BFS, adapted for $\mathbf{G_{r-1} = (V, E_{r-1})}$

```
4.12.1  BFS(q, σ_i, σ_t, G_{r-1})
4.12.2  Input : a branch ID, q, where σ_q ∈ V; the current cell, σ_i ∈ V;
4.12.3          the target cell, σ_t ∈ V; the residual digraph, G_{r-1}
4.12.4  if σ_i = σ_t then
4.12.5     if q ∈ σ_t.BranchIDs then return null // cut this search path
4.12.6     σ_t.BranchIDs.Add(q)                 // record this branch ID
4.12.7     return σ_t                           // success
4.12.8  endif
4.12.9  if σ_i is visited then return null      // stop this search path
4.12.10 set σ_i as visited
```

```
4.12.11  β = null
4.12.12  parallel foreach (σ_i, σ_k) ∈ E_{r−1}
4.12.13     β = β ?? BFS(q, σ_k, σ_t, G_{r−1})      // if β! = null, β = β;
                                                     otherwise, β = BFS(q, σ_k, σ_t, G_{r−1})
4.12.14  endfor
4.12.15  if β ≠ null return σ_i.β                    // success chain
4.12.16  return null                                 // stop this search path
4.12.17  Output: one σ_i-to-σ_t path, if any; otherwise, null
```

BFS-Edge-A works in successive search rounds.

Path search: Search round r attempts to find new augmenting paths using BFS, Pseudocode 4.12, on the current residual digraph, $G_{r-1} = (V, E_{r-1})$. Search round r starts when the source cell, σ_s, broadcasts visit tokens. This search is performed by launching multiple parallel processing tasks (lines 4.11.9–11).

Unvisited intermediate cells that receive visit tokens mark themselves as visited and become new frontier cells. Visited intermediate cells ignore all received visit tokens. Current frontier cells send out visit tokens to all its children, implemented as calls to BFS (Pseudocode 4.12).

Note that, this is a *virtual residual digraph*. The actual algorithm needs additional housekeeping to send visit tokens over original arcs to all original children and over reversed original arcs to all disjoint path predecessors, as discussed in the token handling rules in Section 4.5.1.

Progress: Although not shown in the pseudocodes, the advancing frontiers periodically send *progress indicators* back to the source every other step: (a) *extending notifications* (at least one search path is still extending) and (b) *path confirmations* (at least one search path is successful, i.e. a new augmenting path is 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. Thus, lines 4.11.15–16 are actually done within line 4.12.15, during the return from a successful search. If no progress indicator arrives in the expected time, σ_s assumes that the search round ends.

Successful round and reset: If at least one search path is 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 (line 4.11.18).

Failed round and finalisation: Otherwise (if no augmenting path was found) (line 4.11.12), σ_s initiates a global broadcast, carried by a finalise token. This finalisation is not explicit in the pseudocodes and not strictly necessary, which just informs all cells that the algorithm has terminated.

Branch cut technique: 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.

To solve this problem, a branch-cut technique based on branch IDs is used [51, 52]. Given a search path, $\tau = \sigma_s.\sigma_q.\tau'$, its *branch ID* is q, i.e. the cell ID of its *first* search path cell succeeding the source. In a message-based system, these branch IDs piggyback on top of the visit tokens. The target cell, σ_t, collects all received branch IDs, accepts search paths starting with previously unseen branch IDs and rejects the rest (lines 4.12.5–7). The following result is straightforward.

Lemma 4.10. In any search round, search paths that share the same branch ID are incompatible; only one of them can succeed and become an augmenting path.

Example 4.7. Figure 4.9 shows four search paths arriving at the target cell, σ_7: $\pi = \sigma_1.\sigma_2.\sigma_7$, $\tau_1 = \sigma_1.\sigma_2.\sigma_4.\sigma_7$, $\tau_2 = \sigma_1.\sigma_2.\sigma_6.\sigma_7$ and $\tau_3 = \sigma_1.\sigma_3.\sigma_5.\sigma_7$; their branch IDs are 2, 2, 2 and 3 respectively. Assume that π arrives first. Target σ_7 records π's branch ID, 2, and accepts π as an augmenting path.

Next, consider the fate of the other search paths, τ_1, τ_2 and τ_3, which attempt to reach σ_7 in the next step. Paths τ_1 and τ_2 fail, because they share π's recorded branch ID, 2. However, τ_3 succeeds as a new augmenting path, because it has a different branch ID, 3. Briefly, in this scenario, this search round succeeds with two augmenting paths, π and τ_3, and target σ_7 records their branch IDs, 2 and 3.

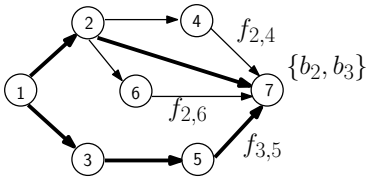


Figure 4.9: Search paths sharing the same branch ID are incompatible. $f_{q,i}$ is the visit token sent by σ_i from a search path with branch ID q and b_q records a branch ID, q.

The above arguments can be summarised by the following result.

Theorem 4.11. Algorithm BFS-Edge-A finds a maximum cardinality set of edge-disjoint paths.

xP Specification 4.5: BFS-Edge-A

Input: All cells start in the same initial state, S_0, with the same set of rules, without any topological awareness (they do not even know their neighbours). 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 state, S_{60}. 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: disjoint path predecessors, d'_j's, and disjoint path successors, d''_k's.

Symbols and states

Cell σ_i uses the following symbols to record its relationships with neighbour cells, σ_j and σ_k:

- n'_j indicates a structural parent;
- n''_k indicates a structural child;
- d'_j indicates a dp-predecessor;
- d''_k indicates a dp-successor;
- s'_j indicates a sp-predecessor (no sp-successor is recorded);
- b_k records the branch ID of a search path for an intermediated cell and records the branch ID of an augmenting path for the target cell.

To implement the branch-cut technique, two kinds of visit tokens are used:

- f_s sent by the source cell, σ_s, and
- $f_{k,i}$ sent by an intermediate cell, σ_i, of branch k.

Also, cell σ_i sends out the following symbols:

- e_i is an extending notification;
- a_i is a path confirmation;
- q is a reset token;
- g is a finalise token.

Cell σ_i uses the following symbols to record its states:

- f indicates that it is the frontier cell;
- d indicates that it appears on a disjoint path;
- specifically, the source cell, σ_s, uses x, y to implement a timer to wait for progress indicators.

Our xP Specification 4.5 uses the same states as xP Specification 4.2.

The matrix R of xP Specification 4.5, $\Pi_1;\Pi_2$, is constructed by sequential composition of Π_1 and Π_2, where Π_1 is SynchNDD (§ 4.2) and Π_2 is the disjoint path search algorithm consisting of nine vectors, informally presented in five groups, according to their functionality and applicability.

Π_1: SynchNDD (see Section 4.2)

Π_2: Disjoint path search

2. Initial differentiation (S_3)

2.1.

2.1.1. $S_3 \rightarrow_{\texttt{min}} S_{10}\ f \mid s$

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

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

3. Source cell (S_{10})

3.1. Receiving progress indicators

3.1.1. $S_{10}\ a_k \rightarrow_{\texttt{min.max}} S_{10} \mid a_k$

3.1.2. $S_{10}\ x\ y\ a_k \rightarrow_{\texttt{min.min}} S_{10}\ d_k\ d\ x \mid d_k$

3.1.3. $S_{10}\ a_k \rightarrow_{\texttt{max.min}} S_{10}\ d''_k\ d \mid \neg\ d''_k$

3.1.4. $S_{10}\ a_k \rightarrow_{\texttt{max.max}} S_{10} \mid d''_k$

3.1.5. $S_{10}\ x\ y\ e_k \rightarrow_{\texttt{min.min}} S_{10}\ x$

3.1.6. $S_{10}\ e_k \rightarrow_{\texttt{max.max}} S_{10}$

3.1.7. $S_{10}\ x\ y \rightarrow_{\texttt{min}} S_{40}(q)\updownarrow \mid d$

3.1.8. $S_{10}\ x\ y \rightarrow_{\texttt{min}} S_{50}(g)\updownarrow \mid \neg\ d$

3.2. Start the search

3.2.1. $S_{10} \rightarrow_{\texttt{min.min}} S_{10}\ x \mid f\ n''_k \neg\ d''_k$

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

3.2.3. $S_{10} \rightarrow_{\texttt{min.min}} S_{10}\ (f_i)\updownarrow_k \mid \iota_i\ f\ n''_k \neg\ d''_k$

3.2.4. $S_{10}\ f \rightarrow_{\texttt{min}} S_{10}$

3.3. Wait

3.3.1. $S_{10}\ x \rightarrow_{\texttt{min}} S_{10}\ x\ y$

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

4. Intermediate cells (S_{20})

4.1.

4.1.1. $S_{20} \rightarrow_{\texttt{min}} S_{40}\ (q)\updownarrow \mid q$

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

4.1.3. $S_{20}\ f_j \rightarrow_{\texttt{min.min}} S_{20}\ f\ v\ s'_j\ b_i\ (e_i)\updownarrow_j \mid \iota_i\ \neg\ v$

4.1.4. $S_{20}\ f_{k,j} \rightarrow_{\texttt{min.min}} S_{20}\ f\ v\ s'_j\ b_k\ (e_i)\updownarrow_j \mid \iota_i\ \neg\ v$

4.1.5. $S_{20}\ f_{k,j} \rightarrow_{\texttt{max.max}} S_{20} \mid v$

4.1.6. $S_{20} \rightarrow_{\texttt{max.min}} S_{20}\ (f_{j,i})\downarrow_k \mid \iota_i\ f\ n''_k\ b_j\ \neg\ d'_k\ d''_k$

4.1.7. $S_{20} \rightarrow_{\texttt{max.min}} S_{20}\ (f_{j,i})\uparrow_k \mid \iota_i\ f\ d'_k\ b_j$

4.1.8. $S_{20}\ f \rightarrow_{\texttt{min}} S_{20}$

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

4.1.10. $S_{20}\ e_k \rightarrow_{\texttt{max.max}} S_{20}$

4.1.11. $S_{20}\ a_k \rightarrow_{\texttt{max.max}} \mid a_k$

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

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

5. Target cell (S_{30})

5.1.

5.1.1. $S_{30} \rightarrow_{\texttt{min}} S_{40}\ (q)\updownarrow \mid q$

5.1.2. $S_{30} \rightarrow_{\texttt{min}} S_{50}\ (g)\updownarrow \mid g$

5.1.3. $S_{30}\ f_j \rightarrow_{\texttt{max.min}} S_{30}\ d'_j\ (a_i)\updownarrow_j$

5.1.4. $S_{30}\ f_{k,j} \rightarrow_{\texttt{max.min}} S_{30}\ d'_j\ b_k\ (a_i)\updownarrow_j\ \neg\ b_k$

5.1.5. $S_{30}\ f_{k,j} \rightarrow_{\texttt{max.min}} S_{30} \mid b_k$

6. All cells (S_{40}, S_{41}, S_{50})

6.1.

6.1.1. $S_{40} \rightarrow_{\texttt{min}} S_{41}$

6.2. End of each search round

6.2.1. $S_{41}\ b_j \rightarrow_{\texttt{max.min}} S_3$

6.2.2. $S_{41}\ s'_j \rightarrow_{\texttt{max.min}} S_3$

6.2.3. $S_{41}\ f_j \rightarrow_{\texttt{max.min}} S_3$

6.2.4. $S_{41}\ f_{k,j} \rightarrow_{\texttt{max.min}} S_3$

6.2.5. $S_{41}\ a_k \rightarrow_{\texttt{max.max}} S_3$

6.2.6. $S_{41}\ d \rightarrow_{\texttt{max}} S_3 \mid s$

6.2.7. $S_{41}\ v\ \rightarrow_{\texttt{max}}\ S_3$

6.2.8. $S_{41}\ q\ \rightarrow_{\texttt{max}}\ S_3$

6.2.9. $S_{41}\ \rightarrow_{\texttt{min}}\ S_3$

6.3. End of the algorithm

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

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

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

6.3.4. $S_{50}\ b_j\ \rightarrow_{\texttt{max.min}}\ S_{50}$

6.3.5. $S_{50}\ s'_j\ \rightarrow_{\texttt{max.min}}\ S_{50}$

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

6.3.7. $S_{50}\ d\ \rightarrow_{\texttt{max}}\ S_{50}$

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

Initial and final configurations

Table 4.11 shows the initial and final configurations of xP Specification 4.5 for Figure 4.1.

Table 4.11: Initial and final configurations of xP Specification 4.2 for Figure 4.1.

Cell	σ_1	σ_2	σ_3	σ_4
Initial	$S_0\ \iota_1\ s$	$S_0\ \iota_2$	$S_0\ \iota_3$	$S_0\ \iota_4$
Final	$S_{60}\ \iota_1\ s\ d''_2\ d''_3\ d''_4$	$S_{60}\ \iota_2\ d'_1\ d''_5$	$S_{60}\ \iota_3\ d'_1\ d''_7$	$S_{60}\ \iota_4\ d'_1\ d''_6$
Cell	σ_5	σ_6	σ_7	σ_8
Initial	$S_0\ \iota_5$	$S_0\ \iota_6$	$S_0\ \iota_7$	$S_0\ \iota_8\ t$
Final	$S_{60}\ \iota_5\ d'_2\ d''_8$	$S_{60}\ \iota_6\ d'_4\ d''_8$	$S_{60}\ \iota_7\ d'_3\ d''_8$	$S_{60}\ \iota_8\ t\ d'_5\ d'_6\ d'_7$

Rule explanations

Path search: At the start of each search round, the source cell, σ_s, broadcasts its visit token, f_s, to all its children that are not its dp-successors, $n''_k\ \neg\ d''_k$, if any (rule 3.2.3), and starts its timer to wait for progress indicators (rule 3.2.1).

An intermediate cell, σ_i, handles received visit tokens using the following rules corresponding to the token handling rules discussed in Section 4.5.1.

(I) Rules 4.1.3–4: (1) when unvisited σ_i, $\neg\ v$, receives f_s from σ_s, it records its sp-predecessor as s'_s and its branch ID as b_i (rule 4.1.3); (2) when unvisited σ_i, $\neg\ v$, receives $f_{k,j}$'s from intermediate cells, it selects *one* of the sending cells by using `min.min` mode as its sp-predecessor, s'_j, and records its branch ID as b_k (rule 4.1.4). In both cases (1) and (2), σ_i marks itself as visited by v, becomes a frontier cell by generating one f and sends back an extending notification, e_i, to its sp-predecessor, σ_s, respectively σ_j.

(a') Rule 4.1.6: σ_i sends $f_{j,i}$, where j is its branch ID, b_j, to all its children that are neither its dp-predecessors nor its dp-successors, $n''_k \neg d'_k \; d''_k$.

(a") Rule 4.1.7: also, σ_i sends $f_{j,i}$, where j is its branch ID, b_j, to all dp-predecessors, d'_k.

(II) Rule 4.1.5: A visited σ_i, v, ignores all received visit tokens, $f_{k,j}$'s.

Progress: An intermediate cell, σ_i, relays progress indicators. On receiving extending notifications, e_k's, cell σ_i forwards one e_i to its sp-predecessor, s'_j (rules 4.1.9–10). On receiving a path confirmation, a_k, cell σ_i records its dp-successor, d''_k, transforms its sp-predecessor, s'_j, into a dp-predecessor, d'_j, and forwards a_i to σ_j (rule 4.1.12). Then, each end of the cancelling arc pair, d'_j and d''_j, are deleted, if any (rule 4.1.13).

The source cell, σ_s, waits one step (rule 3.3.1) for the progress indicators (rules 3.1.2–4 for path confirmations, rules 3.1.5–6 for extending notifications). If no expected progress indicator arrives, then σ_s assumes that the search round ends.

Successful round and reset: If a new augmenting path was found, d, the source cell, σ_s broadcasts a reset token, q (rule 3.1.7). On receiving a reset token, q, σ_i erases v (rules 6.1.1, 6.2.7).

Failed round and finalisation: Otherwise (if no new augmenting path was found), $\neg \; d$, the source cell, σ_s broadcast a finalise token, g, to prompt all cells to clean up useless symbols (rule 3.1.8).

Branch cut technique: The target cell, σ_t, accepts either (1) f_s (rule 5.1.3) or (2) $f_{k,j}$, if $f_{k,j}$ is not from its collected branch IDs, $\neg \; b_k$ (rules 5.1.4–5). In case (2), σ_t records the branch ID as b_k. 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.

Partial traces

Table 4.12 shows cells' contents of xP Specification 4.5 for the five snapshots in Figure 4.1.

4.5.3 Algorithm BFS-Edge-B

Algorithm BFS-Edge-B proposed in my own work [65] builds on BFS-Edge-A and improves it by discarding "dead" cells, which are zero-outdegree cells and the cells, of which all children are conceptually discarded (marked as permanently visited). This pruning is performed in the first search round, parallel with the main search, without affecting the performance.

Algorithm BFS-Edge-B is described by Pseudocodes 4.13 and 4.14 and uses the same variables as BFS-Edge-A, plus a few more specific. Local cell variable *endnode*

Table 4.12: Partial traces of xP Specification 4.5 for Figure 4.1.

Fig.	σ_1	σ_2	σ_3	σ_4
Top.	$S_{10}\ \iota_1\ s\ n''_2 n''_3 n''_4$	$S_{20}\ \iota_2\ n'_1 n''_5$	$S_{20}\ \iota_3\ n'_1 n''_6 n''_7$	$S_{20}\ \iota_4\ n'_1 n''_6$
(a)(b)	$S_{10}\ \iota_1\ s\ d''_2\ d''_3\ \ldots$	$S_{20}\ \iota_2\ d'_1\ d''_5\ \ldots$	$S_{20}\ \iota_3\ d'_1\ d''_6\ \ldots$	$S_{20}\ \iota_4\ \ldots$
(c)	$S_{10}\ \iota_1\ s\ d''_2\ d''_3\ \ldots$	$S_{20}\ \iota_2\ d'_1\ d''_5\ \ldots$	$S_{20}\ \iota_3\ d'_1\ d''_6\ s'_6\ \ldots$	$S_{20}\ \iota_4\ s'_1\ \ldots$
(e)(f)	$S_{10}\ \iota_1\ s\ d''_2\ d''_3\ d''_4$	$S_{20}\ \iota_2\ d'_1\ d''_5$	$S_{20}\ \iota_3\ d'_1\ d''_7$	$S_{20}\ \iota_4\ d'_1\ d''_6$
Fig.	σ_5	σ_6	σ_7	σ_8
Top.	$S_{20}\ \iota_5\ n'_2 n''_8$	$S_{20}\ \iota_6\ n'_3 n'_4 n''_8$	$S_{20}\ \iota_7\ n'_3 n''_8$	$S_{30}\ \iota_8\ t\ n'_5 n'_6 n'_7$
(a)(b)	$S_{20}\ \iota_5\ d'_1\ d''_8\ \ldots$	$S_{20}\ \iota_6\ d'_3\ d''_8\ \ldots$	$S_{20}\ \iota_7\ \ldots$	$S_{30}\ \iota_8\ t\ d'_5\ d'_6\ \ldots$
(c)	$S_{20}\ \iota_5\ d'_2\ d''_8\ \ldots$	$S_{20}\ \iota_6\ d'_3\ d''_8\ s'_4\ \ldots$	$S_{20}\ \iota_7\ s'_3\ \ldots$	$S_{30}\ \iota_8\ t\ d'_5\ d'_6\ d'_7\ \ldots$
(e)(f)	$S_{20}\ \iota_5\ d'_2\ d''_8$	$S_{20}\ \iota_6\ d'_4\ d''_8$	$S_{20}\ \iota_7\ d'_3\ d''_8$	$S_{30}\ \iota_8\ t\ d'_5\ d'_6\ d'_7$

(lines 4.14.5, 4.14.16) indicates that a zero-outdegree cell and *discard_c* (lines 4.14.4, 4.14.20–22) indicates that all children are discarded.

BFS-Edge-B improves BFS-Edge-A by using FastBFS, Pseudocode 4.14. If a frontier cell has no child (line 4.14.16), it discards itself by marking itself and all incoming arcs as *permanently visited* by sending permanently visited notifications to all its parents (lines 4.14.23–27). When all outgoing arcs of a previous frontier cell become permanently visited, this cell discards itself by marking itself and all incoming arcs as permanently visited (lines 4.14.23–27).

Note that, in round 1, the residual digraph is the original digraph and thus a cell sends permanently visited notifications to all its original parents. The discarding process runs in parallel with the main search, along its search path's extending notifications without extra steps, which is not explicit in the pseudocodes.

Pseudocode 4.13: BFS-Edge-B

4.13.1 **Input** : a digraph $G = (V, E)$; a source cell, $\sigma_s \in V$;
4.13.2 a target cell, $\sigma_t \in V$
4.13.3 $P_0 = \emptyset$, $G_0 = G$
4.13.4 $r = 0$
4.13.5 $\sigma_t.BranchIDs = \emptyset$
4.13.6 **while true**
4.13.7 **set** σ_s as *visited*
4.13.8 $Paths = \emptyset$
4.13.9 **parallel foreach** $(\sigma_s, \sigma_q) \in E_{r-1}$
4.13.10 $Paths$.Add(FastBFS$(q, \sigma_q, \sigma_t, G_{r-1}, r)$) // Pseudocode 4.12
4.13.11 **endfor**
4.13.12 **if** $Paths = \emptyset$ **then break**
4.13.13 **foreach** $\beta \in Paths$
4.13.14 $\alpha = \sigma_s.\beta$
4.13.15 $\overline{P_{r-1}} = (\overline{P_{r-1}} \setminus \overline{\alpha}^{-1}) \cup (\overline{\alpha} \setminus \overline{P_{r-1}}^{-1})$
4.13.16 $G_{r-1} = (V, E_{r-1})$, where $E_{r-1} = (E \setminus \overline{P_{r-1}}) \cup \overline{P_{r-1}}^{-1}$
4.13.17 **endfor**

4.13.18 **reset** all *visited* cells to *unvisited*
4.13.19 $r = r + 1$
4.13.20 **endwhile**
4.13.21 **Output** : P_r, which is a maximum cardinality set of edge-disjoint paths

Pseudocode 4.14: FastBFS, adapted for $\mathbf{G_{r-1}} = (\mathbf{V}, \mathbf{E_{r-1}})$

4.14.1 FastBFS$(q, \sigma_i, \sigma_t, G_{r-1}, r)$
4.14.2 **Input** : a branch ID, q, where $\sigma_q \in V$; the current cell, $\sigma_i \in V$;
4.14.3 the target cell, $\sigma_t \in V$; the residual digraph, G_{r-1}; round number, r
4.14.4 $\sigma_i.discard\_c =$ **false**
4.14.5 $\sigma_i.endnode =$ **true**
4.14.6 **if** $\sigma_i = \sigma_t$ **then**
4.14.7 **if** $q \in \sigma_t.BranchIDs$ **then return null** // cut this search path
4.14.8 $\sigma_t.BranchIDs$.Add(q) // record this branch ID
4.14.9 **return** σ_t // success
4.14.10 **endif**
4.14.11 **if** σ_i is *visited* **then return null** // stop this search path
4.14.12 **set** σ_i as *visited*
4.14.13 $\beta =$ **null**
4.14.14 **parallel foreach** *unvisited* arc $(\sigma_i, \sigma_k) \in E_{r-1}$
4.14.15 $\beta = \beta$?? FastBFS$(q, \sigma_k, \sigma_t, G_{r-1}, r)$
// if $\beta \neq null$ then $\beta = \beta$; else $\beta =$ BFS$(q, \sigma_k, \sigma_t, G_{r-1}, r)$
4.14.16 **if** $r = 1$ **then** $\sigma_i.endnode =$ **false** // has a child
4.14.17 **endfor**
4.14.18 **if** $\beta \neq$ **null return** $\sigma_i.\beta$ // success chain
4.14.19 **if** $r = 1$
4.14.20 **if all** $(\sigma_i, \sigma_k) \in E_{r-1}$ are *permanently visited* **then**
4.14.21 $\sigma_i.discard\_c =$ **true** // all outgoing arcs are permanently visited
4.14.22 **endif**
4.14.23 **if** $\sigma_i.endnode$ **or** $\sigma_i.discard\_c$ **then**
4.14.24 **set** σ_i as *permanently visited*
4.14.25 **parallel foreach** $(\sigma_j, \sigma_i) \in E_{r-1}$
4.14.26 **set** (σ_j, σ_i) as *permanently visited*
4.14.27 **endfor**
4.14.28 **endif**
4.14.29 **endif**
4.14.30 **return null** // stop this search path
4.14.31 **Output** : one σ_i-to-σ_t path, if any; otherwise, **null**

Example 4.8. Figure 4.10 shows how the pruning runs in parallel with the main search, along its search path's extending notification, in round 1.

(a) The search starts at the source, σ_1.

(b) The search advances one step; σ_2, σ_3 and σ_4 become the frontier and send back extending notifications, e_2, e_3 and e_4, respectively.

(c) The search advances one step; σ_5 and σ_6 become the frontier and send back extending notifications, e_5 and e_6, respectively; one search path, $\sigma_1.\sigma_4.\sigma_8$, reaches the target, which then sends back a path confirmation, a_8.

The frontier cell, σ_6, has no child, so it discards itself.

(d) The search advances one step; σ_7 becomes the frontier and sends back an extending notification, e_7, to σ_5. The frontier cell, σ_7, has no child, so it discards itself.

(e) Cell σ_5 relays an extending notification, e_5, to σ_2. Because σ_5's only one child, σ_7, is discarded, σ_5 discards itself.

(f) Cell σ_2 relays an extending notification, e_2, to σ_1. Cell σ_2's children, σ_5 and σ_6, are all discarded and σ_3's only one child, σ_6, is discarded, so σ_2 and σ_3 discard themselves.

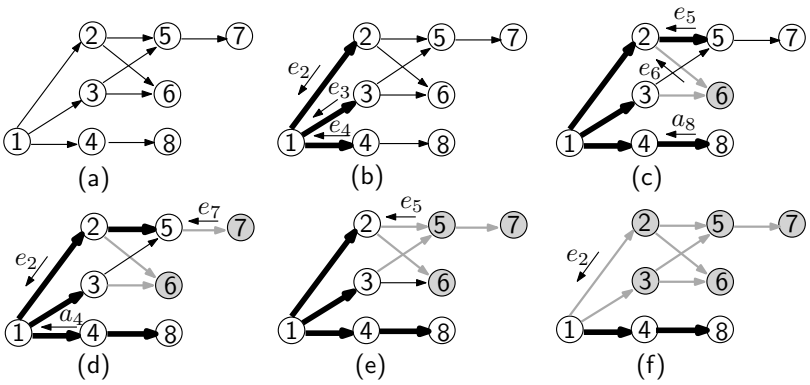


Figure 4.10: The pruning runs in parallel with the main search, along the search path's extending notifications in round 1. Thin arcs: original arcs; thick arcs: search path arcs; gray cells: discarded cells; e_j: an extending notification from cell σ_j; a_j: a path confirmation from cell σ_j.

For BFS-Edge-A, the round 1 search follows the same route as in BFS-Edge-B, without the pruning optimisation. In round 2, because cells σ_2, σ_3, σ_5, σ_6 and σ_7 are not discarded, the search needlessly probes them again. No search path succeeds so the algorithm terminates. Both algorithms terminate in round 2, but BFS-Edge-B is much faster because it does not explore any cell in round 2.

The above arguments can be summarised by the following result.

Theorem 4.12. Algorithm BFS-Edge-B finds a maximum cardinality set of edge-disjoint paths.

xP Specification 4.6: BFS-Edge-B

Input: All cells start in the same initial state, S_0, with the same set of rules, without any topological awareness (they do not even know their neighbours). 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 state, S_{60}. 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: disjoint path predecessors, d'_j's, and disjoint path successors, d''_k's.

Symbols and states

Our xP Specification 4.6 of BFS-Edge-B uses the same symbols and states as xP Specification 4.5 of BFS-Edge-A, plus a few more specific. Cell σ_i uses the following additional symbols:

- o indicates round 1;
- w indicates that it is permanently visited;
- w_i is its permanently visited notification;
- u_k indicates that it is waiting for a permanently visited notification from its child, σ_k.

The matrix R of xP Specification 4.6 is the same as xP Specification 4.5, except that rules 3.2., 4., 6.2. and 6.3. are replaced by the following rules. Moreover, rule 1.1.2 of Π_1 is also replaced as follows: each cell generates one o, which is used to indicate round 1 in Π_2.

1.1.

1.1.2. $S_0\ n\ \rightarrow_{\texttt{min.min}}\ S_1\ o\ (n)\updownarrow\ (n''_i)\uparrow\ (n'_i)\downarrow\ \mid \iota_i$

3.2. Start the search

3.2.1. $S_{10}\ \rightarrow_{\texttt{min.min}}\ S_{10}\ x \mid f\ n''_k \neg d''_k\ w_k$

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

3.2.3. $S_{10}\ \rightarrow_{\texttt{min.min}}\ S_{10}\ (f_i)\updownarrow_k\ \mid \iota_i\ f\ n_k \neg d_k\ w_k$

3.2.4. $S_{10}\ f\ \rightarrow_{\texttt{min}}\ S_{10}$

4. Intermediate cells (S_{20})

4.1.

4.1.1. $S_{20} \rightarrow_{\texttt{min}} S_{40}\ (q)\updownarrow \mid q$

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

4.1.3. $S_{20}\ f_j \rightarrow_{\texttt{min.min}} S_{20}\ f\ v\ s'_j\ b_i\ (e_i)\updownarrow_j \mid \iota_i\ \neg\ v$

4.1.4. $S_{20}\ f_{k,j} \rightarrow_{\texttt{min.min}} S_{20}\ f\ v\ s'_j\ b_k\ (e_i)\updownarrow_j \mid \iota_i\ \neg\ v$

4.1.5. $S_{20}\ f_{k,j} \rightarrow_{\texttt{max.max}} S_{20} \mid v$

4.1.6. $S_{20} \rightarrow_{\texttt{max.min}} S_{20}\ u_k\ (f_{j,i})\downarrow_k \mid \iota_i\ o\ f\ n''_k\ b_j\ \neg\ d'_k\ d''_k\ w_k$

4.1.7. $S_{20} \rightarrow_{\texttt{max.min}} S_{20}\ (f_{j,i})\downarrow_k \mid \iota_i\ f\ n''_k\ b_j\ \neg\ o\ d'_k\ d''_k\ w_k$

4.1.8. $S_{20} \rightarrow_{\texttt{max.min}} S_{20}\ (f_{k,i})\uparrow_k \mid \iota_i\ f\ d'_k\ b_j$

4.1.9. $S_{20}\ f \rightarrow_{\texttt{min}} S_{20}$

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

4.1.11. $S_{20}\ e_k \rightarrow_{\texttt{max.max}} S_{20}$

4.1.12. $S_{20}\ a_k \rightarrow_{\texttt{max.max}} \mid a_k$

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

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

4.1.15. $S_{20}\ w_k\ u_k \rightarrow_{\texttt{max.min}} S_{20}\ o\ w_k \mid n''_k$

4.1.16. $S_{20} \rightarrow_{\texttt{min.min}} S_{20}\ w\ (w_i)\uparrow \mid o\ v\ \neg\ w\ u_k$

6.2. End of each search round

6.2.1. $S_{41}\ b_j \rightarrow_{\texttt{max.min}} S_3$

6.2.2. $S_{41}\ s'_j \rightarrow_{\texttt{max.min}} S_3$

6.2.3. $S_{41}\ f_j \rightarrow_{\texttt{max.min}} S_3$

6.2.4. $S_{41}\ f_{k,j} \rightarrow_{\texttt{max.min}} S_3$

6.2.5. $S_{41}\ a_k \rightarrow_{\texttt{max.max}} S_3$

6.2.6. $S_{41}\ u_k \rightarrow_{\texttt{max.max}} S_3$

6.2.7. $S_{41}\ d \rightarrow_{\texttt{max}} S_3 \mid s$

6.2.8. $S_{41}\ v \rightarrow_{\texttt{max}} S_3$

6.2.9. $S_{41}\ q \rightarrow_{\texttt{max}} S_3$

6.2.10. $S_{41}\ o \rightarrow_{\texttt{max}} S_3$

6.2.11. $S_{41} \rightarrow_{\texttt{min}} S_3$

6.3. End of the algorithm

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

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

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

6.3.4. $S_{50}\ b_j \rightarrow_{\texttt{max.min}} S_{50}$

6.3.5. $S_{50}\ s'_j \rightarrow_{\texttt{max.min}} S_{50}$

6.3.6. $S_{50}\ w_j \rightarrow_{\texttt{max.min}} S_{50}$

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

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

6.3.9. $S_{50}\ d \rightarrow_{\texttt{max}} S_{50}$

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

Initial and final configurations

Table 4.13 shows the initial and final configurations of xP Specification 4.6 for Figure 4.1.

Table 4.13: Initial and final configurations of xP Specification 4.2 for Figure 4.1.

Cell	σ_1	σ_2	σ_3	σ_4
Initial	$S_0\ \iota_1\ s$	$S_0\ \iota_2$	$S_0\ \iota_3$	$S_0\ \iota_4$
Final	$S_{60}\ \iota_1\ s\ d''_1$	$S_{60}\ \iota_2$	$S_{60}\ \iota_3$	$S_{60}\ \iota_4\ d'_1\ d''_8$
Cell	σ_5	σ_6	σ_7	σ_8
Initial	$S_0\ \iota_5$	$S_0\ \iota_6$	$S_0\ \iota_7$	$S_0\ \iota_8$
Final	$S_{60}\ \iota_5$	$S_{60}\ \iota_6$	$S_{60}\ \iota_7$	$S_{60}\ \iota_8\ t\ d'_4$

Rule explanations

Here we only explain the rules for implementing the optimization of BFS-Edge-B.

In round 1, indicated by symbol o, when an intermediate cell, σ_i, sends a visit token to each child σ_k, it generates one u_k (rule 4.1.6). Later, for each received permanently visited notification, w_k, σ_i erases u_k (rule 4.1.15).

An intermediate cell, σ_i, which satisfies one of the following conditions, marks itself as permanently visited by w and sends its permanently visited notification, w_i, to all its parents (rule 4.1.16):

- σ_i is a frontier and it has no child, $\neg\ u_k$ (rule 4.1.16) or
- σ_i is a previous frontier cell and has received permanently visited notifications from all its children, $\neg\ u_k$ (also rule 4.1.16).

Subsequent searches will use a trimmed digraph: in the forward mode, a cell sends its visit token only to children that are not discarded, $\neg\ w_k$ (rule 3.2.3 for the source cell and rules 4.1.6–7 for intermediate cells).

Partial traces

Table 4.14 shows the partial traces of xP Specification 4.6 for cell σ_2 in Figure 4.10. Omitted symbols $(\dots)$ are $\sigma_2\ o\ n'_1\ n''_5\ n''_6$. Highlighted symbols show that the discarding process runs in parallel with the main search, along its search path's extending notifications.

Table 4.14: Partial traces of xP Specification 4.6 for cell σ_2 in Figure 4.10.

Fig.	Evolution	Content
(a)		$\dots$
(b)	$\{f_1\} \Rightarrow v\ s'_1\ b_1\ u_5\ u_6\ \{f_{2,2}\}_{5,6}\ \{e_2\}_1$	$v\ s'_1\ b_1\ u_5\ u_6\ \dots$
(c)		$v\ s'_1\ b_1\ u_5\ u_6\ \dots$
(d)	$\{e_5\ \boldsymbol{e_6}\ \boldsymbol{w_6}\}\ u_6\ w_6 \Rightarrow \{e_2\}_1$	$v\ s'_1\ b_1\ u_5\ w_6\ \dots$
(e)		$v\ s'_1\ b_1\ u_5\ w_6\ \dots$
(f)	$\{\boldsymbol{e_5}\ \boldsymbol{w_5}\}\ u_5 \Rightarrow w\ w_5\ \{\boldsymbol{e_2}\ \boldsymbol{w_2}\}_1$	$v\ s'_1\ b_1\ w_5\ w_6\ w\ \dots$

4.6 BFS-based Node-disjoint Paths Algorithm

This section presents a BFS-based node-disjoint paths solution based on our joint work [51, 52], which is a node-disjoint version of BFS-Edge-A (§ 4.5.2). This algorithm also solves the node-disjoint problem using node-splitting simulation (§ 4.1.2).

4.6.1 Algorithm BFS-Node-A

Algorithm BFS-Node-A [51, 52] is a node-disjoint version of BFS-Edge-A, which solves the node-disjoint problem using node-splitting simulation (§ 4.1.2); the following result is straightforward:

Theorem 4.13. Algorithm BFS-Node-A finds a maximum cardinality set of node-disjoint paths.

xP Specification 4.7: BFS-Node-A

Input: As in the edge-disjoint paths algorithm of xP Specification 4.5.

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

Symbols and states

Our xP Specification 4.7 of BFS-Node-A uses the same symbols and states as xP Specification 4.5 of BFS-Edge-A, plus a few more specific.

Cell σ_i uses the following symbols to record its states:

- $v \to v'$ and v'' indicate its visited entry and exit nodes, respectively;
- c indicates that it must next constrain its in-flow to one.

The rules are the same as the rules of xP Specification 4.5, except that rules 4., 6.2.7 and 6.3.6 are replaced by the following rules.

4. Intermediate cells (S_{20})

4.1.

4.1.1. $S_{20} \ \to_{\texttt{min}} \ S_{40}\ (q)\updownarrow \ \mid q$

4.1.2. $S_{20} \ \to_{\texttt{min}} \ S_{50}\ (g)\updownarrow \ \mid g$

4.1.3. $S_{20}\ f_j \ \to_{\texttt{min.min}} \ S_{20}\ f\ v'\ s'_j\ b_i\ (e_i)\updownarrow_j \ \mid \iota_i \neg\ v'\ v''$

4.1.4. $S_{20}\ f_{k,j} \ \to_{\texttt{min.min}} \ S_{20}\ f\ v''\ s'_j\ b_k\ (e_i)\updownarrow_j \ \mid \iota_i\ d''_j \neg\ v''$

4.1.5. $S_{20}\ f_{k,j} \ \to_{\texttt{min.min}} \ S_{20}\ f\ c\ v'\ s'_j\ b_k\ (e_i)\updownarrow_j \ \mid \iota_i\ d \neg\ v'\ v''\ d''_j$

4.1.6. $S_{20}\ f_{k,j} \ \to_{\texttt{min.min}} \ S_{20}\ f\ v'\ s'_j\ b_k\ (e_i)\updownarrow_j \ \mid \iota_i \neg\ d\ v'\ v''\ d''_j$

4.1.7. $S_{20}\ f_{k,j} \ \to_{\texttt{max.max}} \ S_{20}$

4.1.8. $S_{20} \ \to_{\texttt{min.min}} \ S_{20}\ v'' \mid f\ n''_k \neg\ c\ d'_k\ d''_k\ v''$

4.1.9. $S_{20} \ \to_{\texttt{max.min}} \ S_{20}\ (f_{j,i})\downarrow_k \ \mid \iota_i\ f\ n''_k\ b_j \neg\ c\ d'_k\ d''_k$

4.1.10. $S_{20} \ \to_{\texttt{min.min}} \ S_{20}\ v' \mid f\ d'_k \neg\ v'$

4.1.11. $S_{20} \ \to_{\texttt{max.min}} \ S_{20}\ (f_{j,i})\uparrow_k \ \mid \iota_i\ f\ d'_k\ b_j$

4.1.12. $S_{20}\ f \ \to_{\texttt{min}} \ S_{20}$

4.1.13. $S_{20}\ c \ \to_{\texttt{min}} \ S_{20}$

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

4.1.15. $S_{20}\ e_k \ \to_{\texttt{min.min}} \ S_{20}\ (e_i)\downarrow_j \ \mid \iota_i\ s'_j\ d''_j \neg\ d'_k$

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

4.1.17. $S_{20}\ e_k \ \to_{\texttt{max.max}} \ S_{20}$

4.1.18. $S_{20}\ a_k \ \to_{\texttt{max.max}} \ \mid a_k$

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

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

4.1.21. $S_{20}\ d\ d'_j\ d''_j \ \to_{\texttt{min.min}} \ S_{20}$

6.2. End of each search round

6.2.7.1 $S_{41}\ v' \ \to_{\texttt{max}} \ S_3$

6.2.7.2 $S_{41}\ v'' \ \to_{\texttt{max}} \ S_3$

6.3. End of each search round

6.3.6.1 $S_{50}\ v' \ \to_{\texttt{max}} \ S_{50}$

6.3.6.2 $S_{50}\ v''\ \rightarrow_{\texttt{max}}\ S_{50}$

Initial and final configurations

Table 4.15 shows the initial and final configurations of xP Specification 4.7 for Figure 4.8.

Table 4.15: Initial and final configurations of xP Specification 4.7, for Figure 4.8.

Cell	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
Initial	$S_0\ \iota_1\ s$	$S_0\ \iota_2$	$S_0\ \iota_3$	$S_0\ \iota_4$	$S_0\ \iota_5$	$S_0\ \iota_6$
Final	$S_{60}\ \iota_1\ s\ d''_2\ d''_5$	$S_{60}\ \iota_2\ d'_1\ d''_9$	$S_{60}\ \iota_3\ d'_6\ d''_7$	$S_{60}\ \iota_4\ d'_{10}\ d''_{11}$	$S_{60}\ \iota_5\ d'_1\ d''_6$	$S_{60}\ \iota_6\ d'_5\ d''_3$

Cell	σ_7	σ_8	σ_9	σ_{10}	σ_{11}
Initial	$S_0\ \iota_7$	$S_0\ \iota_8$	$S_0\ \iota_9$	$S_0\ \iota_{10}$	$S_0\ \iota_{11}\ t$
Final	$S_{60}\ \iota_7\ d'_3\ d''_8$	$S_{60}\ \iota_8\ d'_7\ d''_{11}$	$S_{60}\ \iota_9\ d'_2\ d''_{10}$	$S_{60}\ \iota_{10}\ d'_9\ d''_4$	$S_{60}\ \iota_{11}\ t\ d'_4\ d''_8$

Rule explanations

As discussed before, BFS-Node-A is a node-disjoint version of BFS-Edge-A, so we only explain the rules of the node-splitting simulation (§ 4.1.2). An intermediate cell, σ_i, handles its received visit tokens using the following rules corresponding to the token-handling rules in Section 4.5.1, adapted for the node-splitting simulation.

(I) Rules 4.1.3–6.

Forward mode: (1) when cell σ_i, which is unvisited on both its entry and exit nodes, $\neg\ v'\ v''$ (visiting restriction in Section 4.1.2), receives f_s from σ_s, cell σ_i sets its sp-predecessor as s'_s and records its branch ID as b_i (rule 4.1.3); (2) when cell σ_i, which is unvisited on both its entry and exit nodes, $\neg\ v'\ v''$ (visiting restriction in Section 4.1.2), receives visit tokens $f_{k,j}$'s from intermediate cells (in the forward mode, with lower priority than rule 4.1.4 of the push-back mode), σ_i selects *one* of the sending cells as its sp-predecessor, s'_j, and records its branch ID as b_k (rule 4.1.6). In both cases (1) and (2), σ_i marks its entry node as visited by v', generates one f and sends back e_i to its sp-predecessor, σ_s, respectively σ_j. Specifically, if σ_i is on a disjoint path, d, it generates one c (rule 4.1.5).

Push-back mode: when cell σ_i, which is unvisited on its exit node, $\neg\ v''$, receives a visit token, $f_{k,j}$, from its dp-successor (in the push-back mode), d''_j, cell σ_i sets it sp-predecessor as s'_j and records its branch ID as b_k. Also, σ_i marks its exit node as visited by v'', generates one f and sends e_i to σ_j (rule 4.1.4).

(a') Rules 4.1.8–9: if σ_i has no constraint, $\neg\ c$, it sends $f_{j,i}$, where j is its branch ID, b_j, to all its children, n''_k, which are neither its dp-predecessor nor its dp-successor, $\neg\ d'_k\ d''_k$ (rule 4.1.9), and marks its exit node as visited by v'' if it is not marked as visited, $\neg\ v''$ (rule 4.1.8).

(a") Rules 4.1.10–11: also, if σ_i has a dp-predecessor, d'_k, it sends $f_{j,i}$ to σ_k, where j is its branch ID, b_j (rule 4.1.11), and marks its entry node as visited by v' if it is not marked as visited, $\neg\ v'$ (rule 4.1.10).

After sending visit tokens, cell σ_i erases c, if any (rule 4.1.13).

(II) Rule 4.1.7: if conditions of rules 4.1.3–6 are not met (rules are applied in the weak priority order), the received visit tokens are not accepted and σ_i simply ignores them.

BFS-Node-A uses the same strategy as DFS-Node-B (§ 4.4.1) to forward a path confirmation, except that a cell *records* a dp-successor instead of transforming a sp-successor into a dp-successor as in DFS-Node-B (because no sp-successor is recorded).

Consider a cell on a disjoint path receives a path confirmation, a_k.

- According to rule 4.1.19, if cell σ_i has a sp-predecessor of its exit node, $s'_j\ d''_j$, it forwards a_i to σ_j, transforms s'_j into d'_j and records d''_k.
- If the conditions of rule 4.1.19 are not met (rules are applied in the weak priority order), then rule 4.1.20 is considered: if σ_i does not have a sp-predecessor of its exit node, it forwards a_i to its only one sp-predecessor, σ_j, transforms s'_j into d'_j and records d''_k.

A cell that is not on a disjoint path has only one sp-predecessor and thus forwards a path confirmation, also using rule 4.1.20.

Example 4.9. In Figure 4.8(b), the search path, $\sigma_1.\sigma_5.\sigma_6.\sigma_3.\sigma_2.\sigma_9.\sigma_{10}.\sigma_4.\sigma_3.\sigma_7.\sigma_8.\sigma_{11}$, which has visited cell σ_3 twice—once by a forward search on its entry node (rules 4.1.5) and again by a push-back on its exit node (rules 4.1.4), becomes an augmenting path. Cell σ_3 contains $\{d'_2, d''_4, s'_4, s'_6\}$.

(a) On receiving a_7 from σ_7, cell σ_3 forwards a_3 to σ_4 because it has a sp-predecessor of its exit node, $s'_4\ d''_4$ (rule 4.1.19). Also, σ_3 transforms s'_4 into d'_4, records d''_7, and next deletes d'_4 and d''_4, as one end of the cancelling arc (rule 4.1.21). At this time, σ_3 contains $\{d'_2, d''_7, s'_6\}$.

(b) Later, on receiving a_2 from σ_2, σ_3 forwards a_3 to its only one sp-predecessor, s'_6 (rule 4.1.20). Also, σ_3 records d''_2, transforms s'_6 into d'_6, and next deletes d'_2 and d''_2, as one end of the cancelling arc (rule 4.1.21). At this time, σ_3 contains $\{d'_6, d''_7\}$.

Forwarding an extending notification is more complicated, because a cell does not record sp-successors and does not transform sp-predecessors into dp-predecessors after forwarding an extending notification.

Consider a cell on a disjoint path, which has two sp-predecessors and receives an extending notification, e_k.

- According to rule 4.1.14, if e_k comes from σ_i's dp-predecessor, d'_k, i.e. the sp-successor of its entry node, cell σ_i forwards e_i to the sp-predecessor of it entry node, s'_l, which is *not* the sp-predecessor of its exit node, s'_j, i.e. its dp-successor, d''_j.
- If the conditions of rule 4.1.14 are not met, then rule 4.1.15 is considered: if e_k does not come from σ_i's dp-predecessor, $\neg\ d'_k$, then e_k must come from the sp-successor of its exit node and σ_i forwards e_i to the sp-predecessor of its exit node, σ_j, indicated by $s'_j\ d''_j$.

A cell that has only one sp-predecessor applies rule 4.1.16 because it does not meet the conditions of rules 4.1.14–15: σ_i forwards e_i to its only one sp-predecessor, s'_j (rule 4.1.16).

Example 4.10. In Figure 4.8(b), search path $\sigma_1.\sigma_5.\sigma_6.\sigma_3.\sigma_2.\sigma_9.\sigma_{10}.\sigma_4.\sigma_3.\sigma_7.\sigma_8.\sigma_{11}$ becomes an augmenting path. Cell σ_3 contains $\{d'_2, d''_4, s'_4, s'_6\}$ and receives periodical extending notifications (from sp-successors of its entry and exit nodes).

(a) When σ_3 receives e_2 from the sp-successor of its entry node, σ_2 (d'_2), σ_3 forwards e_3 to the sp-predecessor of its entry node, σ_6 (s'_6), which is not the sp-predecessor of its exit node, σ_4 ($s'_4\ d''_4$) (rule 4.1.14).

(b) Later, when σ_3 receives e_7 from σ_7, which is not the sp-successor of its entry node and σ_3 has a sp-predecessor of its exit node, σ_4 ($s'_4\ d''_4$), σ_3 forwards e_3 to σ_4 (rule 4.1.15) .

Partial traces

Table 4.16 shows cells' configurations of xP Specification 4.7 for the three snapshots in Figure 4.8. Omitted symbols (. . .) are cell ID, σ_i, parent pointers, n'_j's, and children pointers, n''_k's, which are the same as in the "Top." row.

4.7 Performance

4.7.1 Runtime Complexity

Consider a digraph with n cells and m arcs, where f_e is the maximum number of edge-disjoint paths, f_n is the maximum number of node-disjoint paths and d is the outdegree of the source cell. For brevity, where the context is clear, we use f for either f_e or f_n.

Dinneen et al. showed that their *graph*-based edge- and node-disjoint algorithms run in $O(mn)$ steps [18]. These modified digraph-based versions, DFS-Edge-A* and DFS-Node-A*, share the same upper-bound, $O(mn)$.

Table 4.16: Partial traces of xP Specification 4.7 for Figure 4.8.

Fig.	σ_1	σ_2	σ_3	σ_4	σ_5	σ_6
Top.	$S_{10}\iota_1 s n''_2 n''_5$	$S_{20}\iota_2 n'_1 n''_3 n''_9$	$S_{20}\iota_3 n'_2 n_6 n''_4 n''_7$	$S_{20}\iota_4 n'_3 n'_{10} n''_{11}$	$S_{20}\iota_5 n'_1 n''_6$	$S_{20}\iota_6 n'_5 n''_3$
(a)	$S_{10} s d''_2 \dots$	$S_{20} d'_1 d''_3 \dots$	$S_{20} d'_2 d''_4 \dots$	$S_{20} d'_3 d''_{11} \dots$	$S_{20} \dots$	$S_{20} \dots$
(b)	$S_{10} s d''_2 \dots$	$S_{20} d'_1 d''_3 s'_3 \dots$	$S_{20} d'_2 d''_4 s'_4 s'_6 \dots$	$S_{20} d'_3 d''_{11} s'_{10} \dots$	$S_{20} s'_1 \dots$	$S_{20} s'_5 \dots$
(c)	$S_{60} s d''_2 d''_5 \dots$	$S_{60} d'_1 d''_9 \dots$	$S_{60} d'_6 d''_7 \dots$	$S_{60} d'_{10} d''_{11} \dots$	$S_{60} d'_1 d''_6 \dots$	$S_{60} d'_5 d''_3 \dots$

Fig.	σ_7	σ_8	σ_9	σ_{10}	σ_{11}
Top.	$S_{20}\iota_7 n'_3 n''_8$	$S_{20}\iota_8 n'_7 n''_{11}$	$S_{20}\iota_9 n'_2 n''_{10}$	$S_{20}\iota_{10} n'_9 n''_4$	$S_{30}\iota_{11} t n'_4 n'_8$
(a)	$S_{20} \dots$	$S_{20} \dots$	$S_{20} \dots$	$S_{20} \dots$	$S_{30} t d'_4 \dots$
(b)	$S_{20} s'_3 \dots$	$S_{20} s'_7 \dots$	$S_{20} s'_2 \dots$	$S_{20} s'_9 \dots$	$S_{30} t d'_4 d'_8 \dots$
(c)	$S_{60} d'_3 d''_8 \dots$	$S_{60} d'_7 d''_{11} \dots$	$S_{60} d'_2 d''_{10} \dots$	$S_{60} d'_9 d''_4 \dots$	$S_{60} t d'_4 d'_8 \dots$

However, a closer inspection shows that this upper-bound is not tight and can be improved. These algorithms reset previously visited nodes (to unvisited) only after successful rounds. Thus, in DFS-Edge-A*, each successful round and its immediately preceding failed rounds (if any) traverse at most m arcs. Thus, DFS-Edge-A* runs in $O(mf_e)$ steps.

Theorem 4.14. Algorithm DFS-Edge-A* runs in $O(mf_e)$ steps.

Similarly, in DFS-Node-A*, each successful round and its immediately preceding failed rounds (if any) traverse at most $m + n - 2$ arcs (the virtual split-node graph has $n - 2$ additional arcs). Thus, DFS-Node-A* runs in $O((m + n)f_n)$ steps or in $O(mf_n)$ steps, under the usual assumption that $n \leq m$.

Theorem 4.15. Algorithm DFS-Node-A* runs in $O(mf_n)$ steps.

In DFS-Edge-B, B*, C, C* and D, the source cell starts d search rounds. In each round, using a Cidon's DFS, visited cells notify their neighbours, so the search does not revisit cells which were visited in the same round and thus completes in at most n steps. As earlier mentioned, all other housekeeping operations are performed in parallel with the main search, so DFS-Edge-B, B*, C, C* and D all run in $O(nd)$ steps. In fact, because DFS-Edge-B and D discard all cells visited in *failed* rounds (which do not find augmenting paths), each successful round and its immediately preceding failed rounds (if any) visit at most n nodes. Thus, DFS-Edge-B and D run in $O(nf_e)$ steps. We conjecture that a similar upperbound can also be found for DFS-Edge-C and C*.

Theorem 4.16. Algorithms DFS-Edge-B and D run in $O(nf_e)$ steps.

Conjecture 4.17. Algorithm DFS-Edge-B* runs in $O(nd)$ steps.

Theorem 4.18. Algorithms DFS-Edge-C and C* run in $O(nd)$ steps.

Similarly, in DFS-Node-B, each successful round and its immediately preceding failed rounds (if any) visit at most $2n-2$ virtually split nodes (the virtual split-node graph has $n-2$ additional nodes). Thus, DFS-Node-B runs in $O(nf_n)$ steps.

Theorem 4.19. Algorithm DFS-Node-B runs in $O(nf_n)$ steps.

In BFS-Edge-A and B, at least one augmenting path is found in each search round, so the number of search rounds is at most f_e. Each round searches for at most n steps. Thus, BFS-Edge-A and B run in $O(nf_e)$.

Theorem 4.20. Algorithms BFS-Edge-A and B run in $O(nf_e)$ steps.

Similarly, in BFS-Node-A, each round searches for $2n-2$ steps (taking into account the virtual node split).

Theorem 4.21. Algorithm BFS-Node-A runs in $O(nf_n)$ steps.

Table 4.17 compares the asymptotic complexity of our algorithms against previous algorithms. Our algorithms score well, because they leverage the potentially unbounded parallelism of an ideal distributed system (such as offered by the P systems framework).

Table 4.17: Asymptotic worst-case runtime complexity comparisons. (?) indicates our conjecture for DFS-Edge-C and C*.

Algorithm	Runtime complexity
DFS-Edge-A and DFS-Node-A (Ford-Fulkerson [24], restricted to unit capacity)	$O(mf)$ steps
Edmonds-Karp (BFS, restricted to unit capacity)	$O(mn)$ steps
DFS-Edge-A* and DFS-Node-A* [18]	$O(mf)$ steps
DFS-Edge-B and DFS-Node-B [52]	$O(nf)$ steps
DFS-Edge-B* [52]	$O(nd)$ steps
DFS-Edge-C [22]	$O(nd)$ steps (?)
DFS-Edge-C* [22]	$O(nd)$ steps (?)
DFS-Edge-D [22]	$O(nf)$ steps
BFS-Edge-A and BFS-Node-A [51, 52]	$O(nf)$ steps
BFS-Edge-B [65]	$O(nf)$ steps

However, a typical DFS or BFS search path does not visit all n cells. In our DFS-based algorithms, each search round detects and discards "dead" cells, therefore decreasing the size of the considered digraph. BFS-based algorithms frequently find several augmenting paths in the same round and thus the number of rounds is smaller than f. Therefore, the expected complexity of our algorithms is probably much less than the indicated upper-bounds; their actual expressions are still *open* problems.

4.7.2 Performance

We substantially extend our performance experiments [52, 22, 65] of edge-disjoint paths algorithms: DFS-Edge-A* of Dinneen et al.'s proposal [18], DFS-Edge-B, B*, C, C* and D of our joint work [51, 52, 22], BFS-Edge-A of our joint work [51, 52] and BFS-Edge-B of my own work [65]. These algorithms are benchmarked using their executable xP specifications, on a series of sets of random digraphs generated by NetworkX [32]. Each set has 30 digraphs with n nodes and m arcs, where $n \in \{100, 200, 300, 400\}$ and $m \in \{250, 500, 1000, 1500, 2000, 2500, 3000\}$.

Table 4.18 shows the average speed-up gain percentages of DFS-Edge-B, B*, C, C*, D, BFS-Edge-A and B over DFS-Edge-A* for each set of random digraphs.

We compute the overall average speed-up of our algorithms over DFS-Edge-A* for all these test cases:

- BFS-based algorithms (85%) are much faster than DFS-based algorithms (55%).
- BFS-Edge-B is the fastest (84.8%) among BFS-based algorithms.
- DFS-Edge-D is the fastest (55.5%) among DFS-based algorithms by combining the ideas of B and C. Note that, although DFS-Edge-C shows the same overall average speed-up as D, for some sets of digraphs (e.g., $n = 100$, $m = 250$ and $n = 300$, $m = 1000$), DFS-Edge-C is marginally slower than D.
- The speed-up gains of DFS-Edge-B (55.2%) and B* (51.3%) show that both Cidon's DFS and Theorem 4.5 contribute to performance improvement.
- DFS-Edge-C (55.5%) and C* (55.3%) are as fast as or even marginally faster than DFS-Edge-B (55.2%).

To better assess the performance improvement of BFS-Edge-B over BFS-Edge-A, we also test these two algorithms on a series of sets of single-source dags generated by NetworkX [32]. Each set has 30 dags with n nodes and m arcs, where $n \in \{100, 200, 300, 400\}$ and $m \in \{250, 500, 1000, 1500, 2000, 2500, 3000\}$. Table 4.19 shows the average speed-up gain percentages of BFS-Edge-B over BFS-Edge-A for each set of random single-source dags.

We also compute the overall average speed-up of BFS-Edge-B over BFS-Edge-A. The average 7.5% performance improvement shows that BFS-Edge-B achieves better speed-up over BFS-Edge-A in single-source dags than in general digraphs, because of its pruning feature.

4.8 Summary

This chapter presents our distributed DFS and BFS-based algorithms for solving the edge and node-disjoint paths problem in digraphs: (1) DFS-Edge-B, C, D and BFS-

Table 4.18: Average speed-up gain percentages of DFS-Edge-B, B*, C, C*, D, BFS-Edge-A and B over DFS-Edge-A* for each set of test cases.

			DFS					BFS	
n	m	f	B	B*	C	C*	D	A	B
100	250	1.6	37.2	34.5	38.9	38.9	39.0	80.5	80.6
	500	3.9	60.1	48.3	60.2	60.0	60.2	93.4	93.4
	1000	9.5	72.7	71.4	72.7	72.4	72.7	98.8	98.8
	1500	12.7	80.0	75.3	80.0	79.6	80.0	99.4	99.4
	2000	18.4	79.9	78.6	79.9	79.8	79.9	99.5	99.5
	2500	22.5	83.1	80.9	83.1	82.8	83.1	99.7	99.7
	3000	27.6	85.7	84.7	85.7	85.6	85.7	99.7	99.7
200	250	0.1	5.3	5.0	5.3	5.3	5.3	34.1	34.2
	500	1.5	43.4	35.2	44.9	44.7	44.9	85.1	86.0
	1000	3.6	59.1	53.5	59.3	59.3	59.3	97.1	97.2
	1500	6.0	69.0	65.2	69.0	69.0	69.0	99.1	99.1
	2000	8.2	77.5	72.5	77.5	77.2	77.5	99.5	99.5
	2500	10.4	79.0	75.6	79.1	78.7	79.1	99.6	99.6
	3000	12.9	80.4	77.6	80.4	80.3	80.4	99.7	99.7
300	250	0	0.4	0.4	0.4	0.4	0.4	11.2	11.2
	500	0.4	20.5	18.3	21.3	20.4	21.3	55.7	56.0
	1000	2.1	50.7	41.8	51.3	51.3	51.4	90.7	90.7
	1500	3.7	62.9	52.0	63.0	63.0	63.0	98.4	98.4
	2000	5.5	62.3	59.2	62.3	62.3	62.3	98.9	98.9
	2500	7.1	72.4	68.2	72.4	72.4	72.4	99.6	99.6
	3000	8.6	75.6	72.8	75.6	75.5	75.6	99.7	99.7
400	250	0	0	0	0	0	0	8.0	8.0
	500	0.1	8.1	7.3	8.7	8.3	8.7	44.6	44.9
	1000	1.7	33.1	27.8	35.0	34.8	35.0	87.1	87.4
	1500	2.4	53.5	49.9	53.9	53.9	53.9	97.9	97.9
	2000	4.2	58.9	55.8	59.1	59.1	59.1	98.4	98.4
	2500	5.3	65.4	59.3	65.4	65.4	65.4	99.0	99.0
	3000	6.0	69.0	65.4	69.0	69.0	69.0	99.3	99.3
Overall average			55.2	51.3	55.5	55.3	55.5	84.7	84.8

Table 4.19: Average speed-up gain percentages of BFS-Edge-B over BFS-Edge-A for each set of test cases.

n \ m	250	500	1000	1500	2000	2500	3000
100	7.3	6.5	5.4	6.9	7.7	11.6	11.7
200	3.2	13.5	10.2	5.2	4.0	3.8	3.1
300	0	7.2	14.8	11.9	5.4	6.0	6.0
400	0	2.4	13.4	18.9	11.5	7.6	5.2

Table 4.20: Comparisons of xP system sizes and pseudocode sizes of disjoint paths algorithms.

Algorithm	xP system size	Pseudocode size
DFS-Edge-B	44	39
DFS-Edge-D	79	85
BFS-Edge-A	53	38
BFS-Edge-B	60	52

Edge-A and B for the edge-disjoint paths problem, (2) DFS-Node-B and BFS-Node-A for the node-disjoint paths problem.

Experimentally, on a series of random digraphs, all our edge-disjoint algorithms show significant speed-up over Dinneen et al.'s basic version, DFS-Edge-A* [18]. BFS-Edge-A and B show the most significant improvement because of their inherent parallelism. Interestingly, despite using a different idea, DFS-Edge-B, C, C* and D seem to have a similar performance; in fact, on purely random digraphs, DFS-Edge-C and D seem to be marginally faster than DFS-Edge-B.

All our xP specifications are the direct implementation of the algorithms, so the runtime complexities are the same as the algorithms. Table 4.20 compares the xP system sizes and pseudocode sizes of disjoint paths algorithms.

The xP system sizes (§1.2) are slightly larger than the pseudocodes sizes (§1.1). These size differences are caused by our sequentialised pseudocodes and structural parallel pseudocodes, which use mainly global variable for the purpose of readability, therefore ignoring many distributed details. We can still see that, even at the detailed executable level, xP specifications are comparable in program size with high-level pseudocodes, which ignore a lot of critical details.

Chapter 5

Minimum Spanning Tree Problem

Minimum spanning tree (MST) problem is one of the most challenging problems in distributed computing [42, 34, 23]. It finds a minimum-weight spanning tree in a weighted undirected graph and has many important applications in diverse areas. MST is used in network optimisation, which aims to minimise the total cost for broadcasting in the network. MST is also used in image segmentation [66] and provides a way to detect clusters with irregular boundaries [30]. MST problem can be transformed to solve NP-hard problems, such as the travelling salesman problem and Steiner tree [16]. Moreover, MST is applied in the field of microbiology, e.g., molecular epidemiology [59].

Following my own work [64], this chapter discusses a synchronous MST algorithm, SynchGHS, which is one of the best-known distributed algorithms. We explain its synchronisation strategies and present our xP solution, which is the first solution of the MST problem in P systems.

This chapter is organised in the following way. Section 5.1 describes the MST problem and discusses the high-level *structural parallel* pseudocodes of SynchGHS and its synchronisation barriers in detail. Section 5.2 discusses the challenges in our xP solution, presents its xP specification and analyses its runtime complexity. Finally, Section 5.3 summarises this chapter and compares our xP specification with SynchGHS in terms of the runtime complexity and program size.

5.1 Minimum Spanning Tree in Graphs

MST problem has been intensely studied and three classical algorithms are: Boruvka's algorithm [5], which is the first proposed algorithm, Prim's algorithm [58] and Kruscal's algorithm [40]. These algorithms are all greedy algorithms that run in polynomial time. Boruvka's algorithm has more distributed and parallel potential, while Prim's algorithm and Kruscal's algorithm are not readily applied in the distributed

environment because they process one node or one edge at a time. A number of distributed MST algorithms have been studied; the GHS algorithms [62, 42] proposed by Gallager, Humblet and Spira [28] are based on Boruvka's algorithm. The GHS algorithms include a synchronous version and an asynchronous version; the SynchGHS algorithm is its synchronous version [42]. This section presents the first P solution of SynchGHS based on Lynch's book [42].

5.1.1 Minimum Spanning Trees in Graphs

Given an *undirected* graph, $G = (V, E)$, a *spanning forest* is a forest, i.e. a graph that is acyclic but not necessarily connected, which consists of all nodes in V and a subset of edges in E [42]. A *spanning tree* of G is a spanning forest of G that is connected; with each edge assigned a *distinct* weight, a *minimum spanning tree* (MST) is the minimum-weight spanning tree, which is *unique* [42].

Theorem 5.1. If all edges of a graph G have distinct weights, then there is exactly one MST for G [42].

A *component* is a tree in the spanning forest. An *outgoing* edge is an edge with two endpoints in different components. A *minimum-weight outgoing edge* (MWOE) of a component has the minimum weight of all outgoing edges of the entire component. Components are combined along MWOEs. For each group of components combined along MWOEs, there is a *unique symmetric* MWOE that is the common MWOE of both endpoint components [42]; all other MWOEs are *asymmetric*.

Each component has one *leader*, σ_j, where its cell ID, j, is used as the *component ID*, also called the *leader ID*. The leader of a new component (constructed by combining a group of components) is selected as the cell with lower cell ID, adjacent to the symmetric MWOE.

In Figure 5.1, (b) shows an undirected graph and its spanning forest consisting of three components; (c) shows the minimum spanning tree after combining these three components along their MWOEs, {1,5} and {2,4}, where {1,5} is an asymmetric MWOE, {2,4} is a symmetric MWOE and σ_2 is selected as the new leader.

5.1.2 The SynchGHS Algorithm

Given an undirected graph, $G = (V, E)$, where V is a finite set of *cells*, $V = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, and E is a set of *edges*. The SynchGHS algorithm assumes that:

- edge weights are distinct (if not, we can use a modified version of SynchGHS in Lynch's book [42]);
- each cell has an ID;

- each cell knows its incident edges and their edge weights; and
- each cell knows n, where $n = |V|$ (the total number of cells in the graph).

SynchGHS computes a minimum spanning tree by merging components of a spanning forest, in *synchronisation* levels, as shown in Pseudocode 5.1. To improve readability, the pseudocodes of this distributed algorithm are presented in *structural parallel* versions.

Initially, each cell, σ_i, is the leader of a single-cell component, $\{\sigma_i\}$ (lines 5.1.4–8). The algorithm repeatedly does the following process: in each level, each component, coordinated by its leader (lines 5.1.11–12), searches for its MWOE (lines 5.1.13–16); if at least one MWOE is found, components are combined along MWOEs (lines 5.1.17–20). This process is repeated until no MWOE can be found in the current level (line 5.1.23).

Each level $k > 1$ is *synchronised* to proceed in predetermined $O(n)$ steps: this is the reason why each cell needs to know n.

Our pseudocodes use the following *global* variables: $G = (V, E)$ is the weighted undirected graph; k is a level number; $found$ indicates that at least one MWOE is found in the current level. These global variables are used here for readability but not used in the distributed version. The evolving spanning forest is only implemented via distributed *local* cell variables: *leader* records a cell's leader ID; *best* records the other endpoint of a cell's best edge towards the MWOE; *parent* records a cell's parent; *children* records a cell's children; *results* is a collection of all *convergecast weight results* of a cell; *localmin* records a cell's local minimum weight (for the leader, it records the MWOE's weight);

Synchronisation barriers enable multiple parallel tasks to wait until all tasks have reached a particular point of execution; our pseudocodes use three synchronisation barriers by **wait** operator to ensure that each *phase* in a level proceeds in n steps.

Pseudocode 5.1: SynchGHS

```
5.1.1  Input : an undirected graph G = (V, E)
5.1.2  k = 0
5.1.3  found = false
5.1.4  parallel foreach σ_i ∈ V
5.1.5    σ_i.leader = σ_i
5.1.6    σ_i.parent = null
5.1.7    σ_i.children = ∅
5.1.8  endfor
5.1.9  repeat
5.1.10   k = k + 1
5.1.11   parallel foreach σ_i ∈ V
5.1.12     if (σ_i.leader = σ_i) then
5.1.13       parallel fork
```

```
5.1.14            (σ_i.localmin, σ_i.best) = FindMWOE(σ_i, σ_i, 0)  // Pseudocode 5.2
5.1.15            if (k > 1) then wait 2n          // synchronisation barrier
5.1.16          endfork
5.1.17          if σ_i.localmin < ∞ then
5.1.18            found = true
5.1.19            Combine(σ_i, 0)                          // Pseudocode 5.4
5.1.20          endif
5.1.21        endif
5.1.22      endfor
5.1.23    until found =false
5.1.24    Output: A minimum spanning tree
```

Example 5.1. Figure 5.1 illustrates the MST computation in SynchGHS, showing the resulting forest for each level.

(a) In level 0, each cell is a component and the component leader.

(b) In level 1, cells σ_1 and σ_2 are combined along symmetric MWOE $\{\sigma_1, \sigma_2\}$, obtaining component $C_1 = \{\sigma_1, \sigma_2\}$ with σ_1 as the new leader; σ_4 and σ_6 are combined along symmetric MWOE $\{\sigma_4, \sigma_6\}$, obtaining component $C_4 = \{\sigma_4, \sigma_6\}$ with σ_4 as the new leader; σ_3, σ_5 and σ_7 are combined along symmetric MWOE $\{\sigma_3, \sigma_7\}$ and asymmetric MWOE $\{\sigma_5, \sigma_7\}$, obtaining component $C_3 = \{\sigma_3, \sigma_7, \sigma_5\}$ with σ_3 as the new leader.

(c) In level 2, components C_1, C_4 and C_3 are combined along symmetric MWOE $\{\sigma_2, \sigma_4\}$ and asymmetric MWOE $\{\sigma_1, \sigma_5\}$, obtaining component $C_2' = \{\sigma_6, \sigma_4, \sigma_2, \sigma_1, \sigma_5, \sigma_7, \sigma_3\}$ with σ_2 as the new leader.

(d) In level 3, no MWOE can be found, so the algorithm terminates with a MST rooted at σ_2.

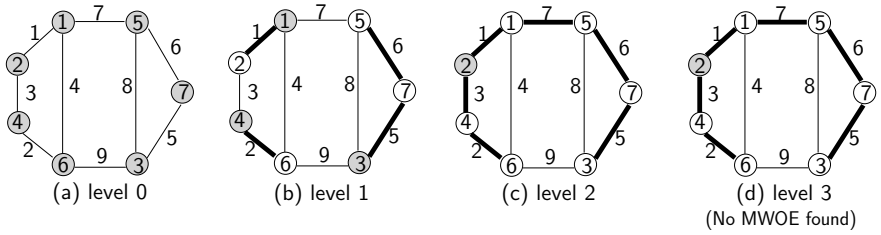


Figure 5.1: MST computation in SynchGHS. Thin edges: non-tree edges; thick edges: tree edges; gray cells: leaders; numbers beside edges: edge weights.

Pseudocodes 5.2 and 5.4 show the details of searching MWOEs and combining components.

Pseudocode 5.2: FindMWOE

```
5.2.1  FindMWOE(σ_i, leader, n_i)
5.2.2  Input : the current cell, σ_i ∈ V, the component leader, leader, and
5.2.3            a counter, n_i, which is the hop count from the leader
5.2.4  σ_i.leader = leader
5.2.5  σ_i.localmin = ∞
5.2.6  σ_i.best = null
5.2.7  σ_i.results = ∅
5.2.8  parallel fork
5.2.9    parallel foreach σ_j ∈ σ_i.children                    // a tree edge
5.2.10     σ_i.results.Add(FindMWOE(σ_j, leader, n_i + 1))
5.2.11   endfor
5.2.12   begin
5.2.13     if (k > 1) then wait n − 1 − n_i        // synchronisation barrier
5.2.14     parallel foreach σ_j ∉ σ_i.children and {σ_i, σ_j} ∈ E//non-tree
5.2.15       if Test(σ_i, σ_j) = true then                    // Pseudocode 5.3
5.2.16         σ_i.results.Add({σ_i, σ_j}.weight, σ_j)
5.2.17       endif
5.2.18     endfor
5.2.19   end
5.2.20 endfork
5.2.21 LocalMin(σ_i)                                          // Pseudocode 5.6
5.2.22 return (σ_i.localmin, σ_i)
5.2.23 Output : a report with the local minimum weight of σ_i
```

Pseudocode 5.3: Test

```
5.3.1  Test(σ_i, σ_j)
5.3.2  Input : cells σ_i, σ_j, where {σ_i, σ_j} ∈ E
5.3.3  if σ_i.leader ≠ σ_j.leader then
5.3.4    return true
5.3.5  else
5.3.6    return false
5.3.7  endif
5.3.8  Output : true if σ_i and σ_k have different leader IDs; otherwise, false
```

Pseudocode 5.4: Combine

```
5.4.1  Combine(σ_i, n_i)
5.4.2  Input : the current cell, σ_i ∈ V, and
5.4.3            a counter, n_i, which is the hop count from the leader
5.4.4  if σ_i.best ∈ σ_i.children then                         // a tree edge
5.4.5    σ_i.children.Remove(σ_i.best)
5.4.6    σ_i.parent = σ_i.best
```

```
5.4.7     σ_i.best.children.Add(σ_i)
5.4.8     Combine(σ_i.best, n_i + 1)
5.4.9   else                                              // MWOE
5.4.10    if (k > 1) then wait n − 1 − n_i      // synchronisation barrier
5.4.11    Connect(σ_i)                                    // Pseudocode 5.5
5.4.12  endif
5.4.13  Output : a new component
```

Pseudocode 5.5: Connect

```
5.5.1   Connect(σ_i)
5.5.2   Input : the current cell, σ_i ∈ V
5.5.3   if σ_i.best.best = σ_i then                       // symmetric MWOE
5.5.4     newLeader = Min(σ_i, σ_i.best)                  // Pseudocode 5.7
5.5.5     newLeader.leader = newLeader
5.5.6     newLeader.child = Max(σ_i, σ_i.best)            // Pseudocode 5.8
5.5.7     Max(σ_i, σ_i.best).parent = newLeader
5.5.8   else                                              // asymmetric MWOE
5.5.9     σ_i.parent = σ_i.best
5.5.10    σ_i.best.children.Add(σ_i)
5.5.11  endif
5.5.12  return
5.5.13  Output : a new leader, newLeader
```

Pseudocode 5.6: LocalMin

```
5.6.1   LocalMin(σ_i)
5.6.2   Input : the current cell, σ_i ∈ V
5.6.3   if σ_i.results = ∅ then return
5.6.4   foreach (weight, σ_j) ∈ σ_i.results
5.6.5     if σ_i.localmin > weight then
5.6.6       σ_i.localmin = weight
5.6.7       σ_i.best = σ_j
5.6.8     endif
5.6.9   endfor
5.6.10  endif
5.6.11  Output : the local minimum weight of cell σ_i
```

Pseudocode 5.7: MinID

```
5.7.1   MinID(σ_i, σ_j)
5.7.2   Input : cells σ_i, σ_j, where {σ_i, σ_j} ∈ E
5.7.3   if i ≤ j then
5.7.4     return σ_i
5.7.5   else
```

```
5.7.6    return σ_j
5.7.7  endif
5.7.8  Output : the cell with lower cell ID
```

Pseudocode 5.8: MaxID

```
5.8.1  MaxID(σ_i, σ_j)
5.8.2  Input : cells σ_i, σ_j, where {σ_i, σ_j} ∈ E
5.8.3  if i ≥ j then
5.8.4    return σ_i
5.8.5  else
5.8.6    return σ_j
5.8.7  endif
5.8.8  Output : the cell with higher cell ID
```

Initially, in level 0, each individual cell is the leader of a single-cell component (lines 5.1.4–8). Inductively, assume that level k components are determined. We now consider level $k+1$ (line 5.1.10).

There are three phases in level $k+1$: (1) *broadcast* phase, (2) *convergecast* phase and (3) *combine* phase, where each phase takes n steps. Specifically, level 1 has no broadcast and convergecast phases because each cell directly selects its minimum-weight incident edge as its MWOE; the last level has no combine phase because no MWOE is found.

The broadcast and converagecast phases are described by functions FindMWOE and Test:

- calls to FindMWOE are implemented by sending *find* messages;
- returns from FindMWOE are implemented by sending *report* messages;
- calls to Test are implemented by sending *test* messages;
- returns from Test are implemented by sending *accept* or *reject* messages.

The combine phase is described by functions Combine and Connect:

- calls to Combine are implemented by sending *change-root* messages;
- calls to Connect are implemented by sending *connect* messages.

All other functions, LocalMin, MinID and MaxID, run locally in the current cell.

Broadcast phase (n steps):

(I) The leader of each component broadcasts a *find* message to propagate its leader ID via tree edges (lines 5.1.14, 5.2.10).

(II) Each receiving cell sends *test* messages including its leader ID via its non-tree edges, to check whether the other endpoint is in the same component (line 5.2.15).

Subphase (I) takes predetermined $n - 1$ steps and (II) takes one step: the synchronisation barrier on lines 5.2.10 and 5.2.13 ensures that all cells have received *find* messages before sending the *test* messages and thus we can check whether an edge is outgoing based on its endpoints' up-to-date leader IDs.

Convergecast phase (n steps):

(I) On receiving a *test* message, a cell sends back an *accept* message if the received leader ID is different from its leader ID (lines 5.3.3–4); otherwise, it sends back a *reject* message (lines 5.3.5–6).

(II) When a cell receives all convergecast weight results, i.e. *accept* and *reject* messages from non-tree edges and *report* messages from tree edges (lines 5.2.8–20), it computes its *local minimum weight*, records its best edge (line 5.2.21) and convergecasts its local minimum weight towards the leader using a *report* message (line 5.2.23). Finally, each leader obtains the MWOE result of the whole component:

(a) if a leader finds a MWOE (line 5.1.17), it starts to combine components (line 5.1.19);

(b) otherwise, if no more MWOE can be found in all components, the algorithm terminates (line 5.1.23). Although it is not explicit in pseudocodes, the leader informs all cells of the algorithm termination by a finalisation broadcast.

Subphase (I) takes one step and (II) takes predetermined $n - 1$ steps: the synchronisation barrier on line 5.1.15 starts a counter when the broadcast phase begins and counts $2n$ steps; the broadcast phase takes n steps and thus this synchronisation barrier ensures that the convergecast phase takes $2n - n = n$ steps and the combine phase starts after all leaders finish searching MWOEs (i.e. after the convergecast phase finishes).

Combine phase (n steps):

(I) Each leader sends a *change-root* message along the best edges towards the cell adjacent to the MWOE; each receiving cell changes its parent pointer in this direction (towards the MWOE) (lines 5.4.4–8).

(II) Cells adjacent to a MWOE send *connect* messages over the MWOE (line 5.4.11). Then, new leaders are selected in the following way:

(a) if *connect* messages are sent both ways along a MWOE, which in this case is symmetric (line 5.5.3), then the endpoint of the MWOE with lower ID becomes the new leader and the other endpoint of the MWOE changes its parent pointer to the new leader (lines 5.5.4–7);

(b) otherwise, if a cell sends a *connect* message along an asymmetric MWOE but does not receive one, it changes its parent pointer to the other endpoint of the MWOE and the other endpoint adds this cell as its child (lines 5.5.8–11).

Subphase (I) takes $n-1$ steps and (II) takes one step: the synchronisation barrier on line 5.4.10 ensures that new leaders are selected after all *connect* messages are sent. Thus, there is no such case that a cell mistakenly thinks it is one endpoint of an asymmetric MWOE because the other *connect* message has not yet been sent.

When the combining of level $k+1$ components finishes, the level $k+2$ components are obtained.

Remark: The solution given here shows one of the various ways to synchronise levels. There are several other ways to synchronise levels, e.g., one may consider predetermined $n-1$ steps for a cell to test its non-tree edges one by one.

Note that levels 0 and 1 do not need to be synchronised (lines 5.1.15, 5.2.12, 5.4.10): in level 0, each cell is a component; in level 1, each cell directly sends a *connect* message along its minimum-weight incident edge.

Using our synchronisation barriers, each synchronisation level k ($k > 1$) starts at step l_k and ends at step $l_k + 3n$, taking totally $3n$ steps.

Example 5.2. Figure 5.2 shows an example of level 2 synchronisation for Figure 5.1 (b), which takes totally 21 steps.

(a) At step $l_2 = 2$, level 2 leaders are selected and each leader starts a *find* message broadcast.

(b) The broadcast of *find* messages continues.

(c) At step $l_2 + n - 1 = 8$, all *test* messages are sent.

(d) At step $l_2 + n = 9$, all *accept* and *reject* messages are sent.

(e) At step $l_2 + n + 1 = 10$, the convergecast of *report* messages starts.

(f) The convergecast of *report* messages continues.

(g) At step $l_2 + 2n = 16$, each leader sends a *change-root* message towards its MWOE.

(h) The forwarding of *change-root* messages continues.

(i) At step $l_2 + 3n - 1 = 22$, all *connect* messages are sent along MWOEs.

(j) At step $l_3 = l_2 + 3n = 23$, new leaders are selected and level 3 starts.

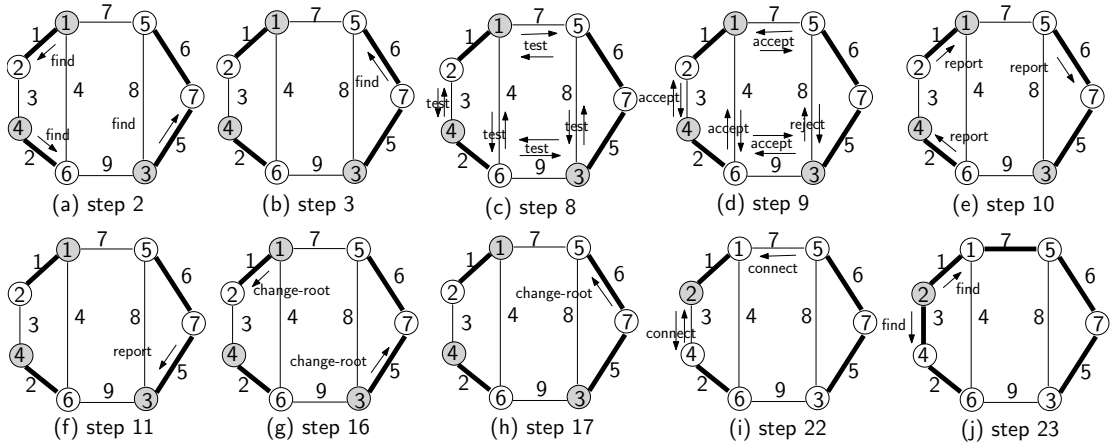


Figure 5.2: An example of level 2 synchronisations in SynchGHS for Figure 5.1 (b). Thin edges: non-tree edges; thick edges: tree edges; gray cells: leaders; numbers beside edges: edge weights; arrows beside edges: messages.

5.2 xP Solution for the MST Problem

As mentioned before, our P solution assumes that each cell knows its incident edges, edge weights and the total number of cells. The P specification has challenging tasks:

- to complete all important details ignored by the *structural parallel* pseudocodes, without increasing the runtime complexity;
- to indicate
  - how to represent weighted edges,
  - how to build and navigate over components,
  - how to implement synchronisation barriers,
  - how to compute the minimum weight, and
  - how to select leaders.

It is critical to compute the minimum value over an arbitrary long sequence in one step (see § 2.3.5 for a rule snippet example): determining the minimum value in arbitrary steps (as in classical P systems) induces asynchronicity, which makes it hard to synchronise.

To implement three synchronisation barriers, three incrementing counters are used: (1) a *broadcast counter* used in *all cells* during the broadcast phase, (2) a *broadcast+convergecast counter* used in *all leaders* during the broadcast and convergecast phases and (3) a *combine counter* used in the *cells along best edges* to MWOEs during the combine phase. The broadcast and combine counters start from 0 and stop at $n-2$; the broadcast+convergecast counter starts from 0 and stops at $2n-1$.

At the start of the broadcast phase, each leader sets its broadcast and broadcast+convergecast counters to 0. The broadcast counter is sent together with *find* messages to all cells: each receiving cell increments the counter by one before broadcasting it. Each cell keeps incrementing its broadcast counter by one; when its counter reaches $n-2$, it stops the counter and sends *test* messages.

Each leader keeps incrementing its broadcast+convergecast counter until it reaches $2n-1$.

One step after the broadcast+convergecast counter stops, each leader sets its combine counter to 0 and the combine phase starts. The combine counter is forwarded together with *change-root* messages towards the cell adjacent to the MWOE: each receiving cell increments the combine counter by one before forwarding it. The cell adjacent to MWOE keeps incrementing the counter; when its counter reaches $n-2$, it stops the counter and sends a *connect* message over the MWOE.

xP Specification 5.1: SynchGHS

Input: All cells start with the same set of rules and the same initial state, S_0. Each cell, σ_i, contains an immutable cell ID symbol, ι_i, a number indicator, $z(c^n)$, where n is the number of cells, and a list of its incident weighted edge symbols, $e_j(c^l)$, where j is the other endpoint cell ID of the edge and l is its edge weight.

Output: All cells end in the same state, S_2. On completion, each cell still contains a number indicator and a list of incident edge symbols. Also, each cell contains MST symbols: tree parent, p_j, tree children, s_k's, and the leader, q_l.

Symbols and states

Cell σ_i uses the following symbols to record its relationships with neighbour cells, σ_j and σ_k:

- p_j indicates a tree parent;
- s_k indicates a tree child;
- y_j records its best edge;
- q_j indicates its leader ID (component ID);

- $e_j(c^l)$ indicates an incident edge, where j is the other endpoint cell ID and l is its edge weight.

Cell σ_i sends out the following symbols:

- f_j is a *find* message, where j is the leader ID;
- $t_{j,i}$ is a *test* message, where j is the leader ID;
- a_i is an *accept* message;
- x_i is a *reject* message or a *report* message indicating that no MOWE is found by σ_i;
- $r_i(c^l)$ is a *report* message indicating the local minimum weight of σ_i;
- h_i is a *change-root* message;
- d_i is a *connect* message;
- g is a finalise token.

Cell σ_i uses the following symbols to compute its local minimum weight:

- w is used for its counter that waits for convergecast weight results;
- $m_j(c^l)$ records the *report* or *accept* result from child σ_j;
- $m'_j(c^l)$ is the intermediate or final local minimum weight result.

Cell σ_i uses the following symbols to indicate its states:

- m' indicates that it is receiving an *accept*, *reject* or *report* message;
- x' indicates that it must next send a *report* message that no MWOE is found by σ_i;
- d indicates that it has sent a connect message.

Cell σ_i uses the following symbols for its counters; all of them start from 1 (they are initialised as 0 but incremented by 1 in the same step), which function as previously discussed:

- $u(c^q)$ is a broadcast counter that counts from 1 to $n-1$ (rules 2.1.3, 2.1.13, 2.1.15);
- $v(c^q)$ is a broadcast+convergecast counter that counts from 1 to $2n$ (rules 2.1.3, 2.1.14, 2.2.17);

- $b(c^q)$ is a combine counter that counts from 1 to $n-1$ (rules 2.3.1–2, 2.3.6–7).

For cell σ_i, a tree edge is indicated by s_k and a non-tree edge is indicated by $e_k(X) \neg s_k\ p_k$.

State S_1 is a computing state and S_2 is a state after being informed of the algorithm termination.

The matrix R of xP Specification 5.1 consists of four vectors, informally presented in two groups, according to their functionality and applicability. Each vector implements an independent function, performed in one step.

1. Level 0 (S_0)

1.1 First action

1.1.1. $S_0 \rightarrow_{\texttt{min.min}} S_1\ m'_j(X) \mid e_j(X)$

1.1.2. $S_0\ m'_j(XY) \rightarrow_{\texttt{max.min}} S_1\ m'_j(X) \mid e_j(X)$

1.1.3. $S_0\ m'_j(X) \rightarrow_{\texttt{min.min}} S_1\ d\ y_j\ (d_i)\updownarrow_j \mid \iota_i$

2. Level $k > 0$ (S_1)

2.1 Leader selection and broadcast phase by *initiate* and *test* messages

// Select leaders

2.1.1. $S_1 \rightarrow_{\texttt{min.min}} S_1\ n_i(c^i) \mid d\ d_j\ y_j\ \iota_i$

2.1.2. $S_1 \rightarrow_{\texttt{min.min}} S_1\ n_j(c^j) \mid d\ d_j\ y_j$

2.1.3. $S_1\ n_i(X)\ n_j(XY)\ d\ d_j\ y_j \rightarrow_{\texttt{min.min}} S_1\ f_i\ s_j\ u(\lambda)\ v(\lambda) \mid \iota_i$

2.1.4. $S_1\ n_i(XY)\ n_j(X)\ d\ d_j\ y_j \rightarrow_{\texttt{min.min}} S_1\ p_j \mid \iota_i$

2.1.5. $S_1\ d\ y_j \rightarrow_{\texttt{min.min}} S_1\ p_j \neg d_j$

2.1.6. $S_1\ d_k \rightarrow_{\texttt{max.min}} S_1\ s_k$

// *initiate* and test messages

2.1.7. $S_1\ q_k \rightarrow_{\texttt{min.min}} S_1 \mid f_j$

2.1.8. $S_1\ y_k \rightarrow_{\texttt{min.min}} S_1 \mid f_j$

2.1.9. $S_1 \rightarrow_{\texttt{min.min}} S_1\ q_j \mid f_j$

2.1.10. $S_1 \rightarrow_{\texttt{max.min}} S_1\ w\ (f_j\ u(Xc))\updownarrow_k \mid s_k\ f_j\ u(X)$

2.1.11. $S_1\ f_j \rightarrow_{\texttt{min.min}} S_1$

2.1.12. $S_1 \rightarrow_{\texttt{max.min}} S_1\ w\ (t_{j,i})\updownarrow_k \mid e_k(X)\ q_j\ u(Yc)\ z(Y)\ \iota_i \neg s_k\ p_k$

2.1.13. $S_1\ u(Y) \rightarrow_{\texttt{min}} S_1\ u(Yc) \mid \neg z(Y)$

2.1.14. $S_1\ v(Y) \rightarrow_{\texttt{min.min}} S_1\ v(Yc) \mid q_i\ \iota_i\ u(X)$

2.1.15. $S_1\ u(Y) \rightarrow_{\texttt{min}} \mid z(Y)$

2.2 Convergecast phase by *accept*, *reject* or *report* messages

2.2.1. $S_1\ t_{j,k}\ \rightarrow_{\texttt{max.min}}\ S_1\ (x_i)\updownarrow_k\ \mid q_j\ \iota_i$

2.2.2. $S_1\ t_{j,k}\ \rightarrow_{\texttt{max.min}}\ S_1\ (a_i)\updownarrow_k\ \mid \iota_i\ \neg\ q_j$

2.2.3. $S_1\ w\ a_j\ \rightarrow_{\texttt{max.min}}\ S_1\ m'\ m_j(X) \mid e_j(X)$

2.2.4. $S_1\ w\ x_j\ \rightarrow_{\texttt{max.min}}\ S_1\ m'$

2.2.5. $S_1\ w\ r_j(X)\ \rightarrow_{\texttt{max.min}}\ S_1\ m'\ m_j(X)$

2.2.6. $S_1\ \rightarrow_{\texttt{min.min}}\ S_1\ x' \mid m'\ \neg\ w\ m_j(X)$

2.2.7. $S_1\ \rightarrow_{\texttt{min.min}}\ S_1\ m'_j(X) \mid m'\ m_j(X)\ \neg\ w$

2.2.8. $S_1\ m'_j(XY)\ \rightarrow_{\texttt{max.min}}\ S_1\ m'_j(X) \mid m'\ m_j(X)\ \neg\ w$

2.2.9. $S_1\ x'\ \rightarrow_{\texttt{min.min}}\ S_1\ (x_i)\updownarrow_k\ \mid m'\ p_k\ \iota_i\ \neg\ w$

2.2.10. $S_1\ m'_j(X)\ \rightarrow_{\texttt{min.min}}\ S_1\ y_j\ r_i(X)\updownarrow_k\ \mid m'\ p_k\ \iota_i\ \neg\ w$

2.2.11. $S_1\ m'_j(X)\ \rightarrow_{\texttt{min.min}}\ S_1\ y_j \mid m'\ \neg\ w$

2.2.12. $S_1\ m'\ \rightarrow_{\texttt{max}}\ S_1$

2.2.13. $S_1\ g\ \rightarrow_{\texttt{min.min}}\ S_2\ (g)\updownarrow_j\ \mid s_j\ \neg\ q_i\ \iota_i$

2.2.14. $S_1\ g\ \rightarrow_{\texttt{max}}\ S_2$

2.2.15. $S_1\ \rightarrow_{\texttt{max.min}}\ S_1\ g\ (g)\updownarrow_j\ \mid x'\ s_j\ q_i\ \iota_i$

2.2.16. $S_1\ x'\ v(Y)\ \rightarrow_{\texttt{min.min}}\ S_1 \mid q_i\ \iota_i$

2.2.17. $S_1\ v(Y)\ \rightarrow_{\texttt{min.min}}\ S_1\ v(Yc) \mid q_i\ \iota_i\ \neg\ z(YY)$

2.3 Combine phase by *change-root* and *connect* messages

2.3.1. $S_1\ v(Y)\ \rightarrow_{\texttt{min.min}}\ S_1\ (h_i\ b(\lambda)) \mid q_i\ z(YY)\ \iota_i$

2.3.2. $S_1\ b(Y)\ s_j\ \rightarrow_{\texttt{min.min}}\ S_1\ p_j\ (h_i\ b(Yc))\updownarrow_j\ \mid y_j\ h_k\ \iota_i$

2.3.3. $S_1\ h_k\ p_k\ \rightarrow_{\texttt{min.min}}\ S_1\ s_k$

2.3.4. $S_1\ h_k\ \rightarrow_{\texttt{min.min}}\ S_1$

2.3.5. $S_1\ \rightarrow_{\texttt{min.min}}\ S_1\ d\ (d_i)\updownarrow_j\ \mid y_j\ b(Yc)\ z(Y)\ \iota_i$

2.3.6. $S_1\ b(Y)\ \rightarrow_{\texttt{min}}\ S_1\ b(Yc)\ \neg\ z(Y)$

2.3.7. $S_1\ b(Y)\ \rightarrow_{\texttt{min}}\ S_1 \mid z(Y)$

Initial and final configurations

Table 5.1 shows the initial and final configurations of xP Specification 5.1 for Figure 5.1. Omitted symbols (...) in the final configuration are a number indicator, $z(c^7)$, and the list of incident edge symbols, which are the same as in the initial configuration.

Take cell σ_1's initial configuration for example, $z(c^7)$ indicates that there are totally 7 cells in the graph; $e_2(c)$, $e_5(c^7)$ and $e_6(c^4)$ indicate its incident weighted edges $\{\sigma_1, \sigma_2\}.weight = 1$, $\{\sigma_1, \sigma_5\}.weight = 7$ and $\{\sigma_1, \sigma_6\}.weight = 4$, respectively.

Table 5.1: Initial and final configurations of xP Specification 5.1 for Figure 5.1.

Step	σ_1	σ_2	σ_3	σ_4
0	$S_0\ \iota_1\ z(c^7)$ $e_2(c)\ e_5(c^7)\ e_6(c^4)$	$S_0\ \iota_2\ z(c^7)$ $e_1(c)\ e_4(c^3)$	$S_0\ \iota_3\ z(c^7)$ $e_5(c^8)\ e_6(c^9)\ e_7(c^5)$	$S_0\ \iota_4\ z(c^7)$ $e_2(c^3)\ e_6(c^2)$
39	$S_2\ \iota_1\ q_2\ p_2\ s_5\ \ldots$	$S_2\ \iota_2\ q_2\ s_1\ s_4\ \ldots$	$S_2\ \iota_3\ q_2\ p_7\ \ldots$	$S_2\ \iota_4\ q_2\ p_2\ s_6\ \ldots$

Step	σ_5	σ_6	σ_7
0	$S_0\ \iota_5\ z(c^7)$ $e_1(c^7)\ e_3(c^8)\ e_7(c^6)$	$S_0\ \iota_6\ z(c^7)$ $e_1(c^4)\ e_4(c^2)\ e_3(c^9)$	$S_0\ \iota_7\ z(c^7)$ $e_3(c^5)\ e_5(c^6)$
39	$S_2\ \iota_5\ q_2\ p_1\ s_7\ \ldots$	$S_2\ \iota_6\ q_2\ p_4\ \ldots$	$S_2\ \iota_7\ q_2\ p_5\ s_3\ \ldots$

Rules

The xP rules correspond to broadcast, covergecast and combine phases and their subphases previously discussed in Section 5.1.2.

Broadcast phase

(I) Rules 2.1.7–2.1.11: on receiving a *find* message, f_j, and a broadcast counter, $u(X)$, cell σ_i erases its old leader ID, q_k, and its old best edge, y_k, if any, and records its leader ID as q_j (rules 2.1.7–9). Then it sends f_j and $u(Xc)$ via all tree edges, s_k, and generates one w for each receiver (rule 2.1.10).

(II) Rule 2.1.12: cell σ_i sends $t_{j,i}$, where j is the leader ID, q_j, via all non-tree edges, $e_k(X) \neg s_k\ p_k$, and generates one w for each receiver.

Convergecast phase:

(I) Rules 2.2.1–2: on receiving $t_{j,k}$, cell σ_i sends x_i to σ_k if it has q_j (the same leader ID) (rule 2.2.1); otherwise, σ_i sends a_i to σ_k (rule 2.2.2).

(II) Rules 2.2.3–12: for each received convergecast result, a_j, x_j or $r_j(X)$, cell σ_i deletes one w and generates one m'; specifically, σ_i records results by generating $m_j(X)$ using $a_j\ e_j(X)$ and transforming $r_j(X)$ into $m_j(X)$ (rules 2.2.3–5).

Consider a cell, σ_i, which has received all convergecast weight results, $m' \neg w$. If any $m_j(Z)$ exists, σ_i computes its local minimum weight, $m'_j(X)$ (rules 2.2.7–8) and then records its best edge as y_j and sends $r_j(X)$ to its parent, p_k (rule 2.2.10). Otherwise, σ_i generates one x' (rule 2.2.6) and sends x_i to its parent, p_k (rule 2.2.9). Finally, each leader, σ_l, obtains its MWOE result (rule 2.2.6 for finding no MWOE and rule 2.2.11 for finding a MWOE).

(a) Rules 2.2.13–15: if no MWOE is found, indicated by x' (rule 2.2.15), σ_l broadcasts a finalise token, g, to announce termination (rules 2.2.13–14).

(b) Otherwise, σ_l starts the combine phase.

Combine phase:

(I) Rules 2.3.1–2: each leader σ_l generates $h_l\ b(\lambda)$, as receiving a *change-root* message and a combine counter from itself. Cell σ_i that receives $h_k\ b(Y)$ from σ_k sends $h_i\ b(Yc)$ via its best edge, y_j, and changes the parent and child pointer directions by transforming s_j to p_j and p_k to s_k.

(II) Rule 2.3.5: cell σ_i adjacent to MWOE sends d_i via its best edge to σ_j, indicated by y_j, and generates one d.

(a) Rules 2.1.1–4: if σ_i receives d_j from σ_j (this is a symmetric MWOE), it compares i and j (rules 2.1.1–2):

- if $i \leq j$, σ_i becomes the new leader and prepares to start the broadcast phase by generating f_i, as receiving a *find* message from itself (rule 2.1.3);
- if $i > j$, σ_i sets its tree parent as p_j (rule 2.1.4).

(b) Rules 2.1.5–6: if σ_i does not receive d_j from σ_j (this is an asymmetric MWOE), it sets its parent as p_j (rule 2.1.5) and σ_j adds σ_i as its child (rule 2.1.6).

Rules 2.1.1–6 are placed in the same vector as the broadcast phase rules to achieve maximum performance: once new leaders are selected, the broadcast phase starts at the same step.

Specifically in level 1, each cell σ_i computes its incident minimum-weight edge as its MWOE, $m'_j(X)$ (rules 1.1.1–1.1.2), records its best edge as y_j, sends a *connect* message, d_i, to σ_j and generates one d (rule 1.1.3). Then new leaders are selected as discussed in subphase (II) of the combine phase, obtaining level 2 components.

Partial traces

Table 5.2 shows partial traces of xP Specification 5.1 for cell σ_i in Figure 5.2. Omitted symbols $(\dots)$ are $\iota_1\ z(c^7)\ e_2(c)\ e_5(c^7)\ e_6(c^4)$.

Since our xP Specification 5.1 is a direct SynchGHS implementation, the following theorem is straightforward:

Theorem 5.2. When the evolution of xP Specification 5.1 stops, tree parent and child pointers indicate a minimum spanning tree.

Given a graph with n cells, the runtime complexity of xP Specification 5.1 is discussed as follows.

Proposition 5.3. The evolution of xP Specification 5.1 takes at most $3nL + 3n + 1$ steps, i.e. $O(n \log n)$, where L is the number of synchronisation levels.

Table 5.2: Partial traces of xP Specification 5.1 for cell σ_1 in Figure 5.2.

Fig.	Evolution	Content
(a)	$\{d_2\}\ d\ y_2 \Rightarrow u(c)\ v(c)\ q_1\ s_2\ w\ \{f_1\ u(c)\}_2$	$q_1\ s_2\ w\ u(c)\ v(c)\ldots$
(b)	$u(c)\ v(c) \Rightarrow u(c^2)\ v(c^2)$	$q_1\ s_2\ w\ u(c^2)\ v(c^2)\ldots$
(c)	$u(c^6)\ v(c^6) \Rightarrow v(c^7)\ w^2\ \{t_{1,1}\}_{5,6}$	$q_1\ s_2\ w^3\ v(c^7)\ldots$
(d)	$\{t_{3,5}\ t_{4,6}\}\ v(c^7) \Rightarrow v(c^8)\ \{a_1\}_{5,6}$	$q_1\ s_2\ w^3\ v(c^8)\ldots$
(e)	$\{a_5\ a_6\}\ w^2\ v(c^8) \Rightarrow v(c^9)\ m_5(c^7)\ m_6(c^4)$	$q_1\ s_2\ w\ m_5(c^7)\ m_6(c^4)\ v(c^9)\ldots$
(f)	$\{r_2(c^3)\}\ w\ m_5(c^7)\ m_6(c^4)\ v(c^9) \Rightarrow\ y_2\ v(c^{10})$	$q_1\ s_2\ y_2\ v(c^{10})\ldots$
(g)	$v(c^{14}) \Rightarrow \{b(c)\ h_1\}_2$	$q_1\ p_2\ y_2\ldots$
(h)		$q_1\ p_2\ y_2\ldots$
(i)		$q_1\ p_2\ y_2\ldots$
(j)	$\{d_5\} \Rightarrow s_5$	$q_1\ p_2\ y_2\ s_5\ldots$

Proof. In level 0, each component consists of an individual cell, so it takes zero step. In level 1, each cell directly sends a *connect* message along its minimum-weight incident edge, so it takes one step. In the last level, (1) the broadcast phase takes n steps; (2) the convergecast phase immediately stops when the leader knows that no MWOE is found, which takes at most n steps; and (3) there is no combine phase. Therefore, the last level takes at most $2n$ steps. As previously discussed, each synchronisation level takes $3n$ steps. Thus, L synchronisation levels take $3nL$ steps.

The finalisation broadcast takes at most $n-1$ steps (the depth of a tree is at most $n-1$) and one step for each receiving cell to change its state. Thus, the finalise broadcast takes at most $n-1+1=n$ steps.

Thus, in total, the evolution of xP Specification 5.1 takes at most $1+2n+3nL+n = 3nL+3n+1$ steps, where L is the number of synchronisation levels. □

Lemma 5.4. The number of synchronisation levels, L, is at most $\log n - 1$.

Proof. In each level $k \geq 1$, each component is combined with at least one other component at the same level, so the number of cells in each component of level $k \geq 1$ is at least 2^{k-1}. Thus, the number of levels $k \geq 1$ is at most $\log n + 1$. As previously discussed, level 1 and the last level are not synchronised to take $3n$ steps, so the number of synchronisation levels, L, is at most $\log n + 1 - 2 = \log n - 1$. □

Theorem 5.5. The runtime of xP Specification 5.1 is $O(n \log n)$.

Proof. Following Proposition 5.3 and Lemma 5.4, the evolution of xP Specification 5.1 takes at most $1 + 2n + 3n(\log n - 1) + n = 3n \log n + 1$ steps, i.e. $O(n \log n)$. □

Thus, the runtime of xP Specification 5.1 is the same as the runtime complexity of SynchGHS in Lynch's book [42], i.e. $O(n \log n)$.

5.3 Summary

This chapter presents the first solution in P systems for one of the most challenging distributed problems, the MST problem, which is a direct implementation of SynchGHS based on Lynch's book [42].

Our xP solution implements synchronisation barriers for each synchronisation level and computes the minimum weight in one step to avoid asynchronicity (which can be a problem if determining the minimum value in arbitrary steps, as in classical P systems). Our xP solution successfully implements all critical distributed details, which demonstrates the adequacy of our xP systems in modelling a challenging distributed algorithm.

The informal high-level pseudocodes use totally 108 lines, while a formal description, xP Specification 5.1, only uses 42 rules, i.e. our xP system size (§1.2) is *less than half* of the pseudocode size (§1.1). As discussed before in Theorem 5.5, the runtime complexity of our xP specification is exactly the same as the runtime complexity of SynchGHS, i.e. $O(n \log n)$.

Taking into account that it offers complete specifications, our xP specification compares favourably, both in terms of program size and runtime complexity, with high-level pseudocodes, which ignore many critical details.

Chapter 6

Conclusion

A P system is a parallel and distributed computational model inspired by the structure and interactions of living cells. Essentially, a P system is specified by its membrane structure (in this thesis, a digraph), symbols and rules. Each cell transforms its content symbols and sends messages to its neighbours using formal rules inspired by rewriting systems. The rules of the same cell can be applied in parallel (where possible) and all cells work in parallel. In a sense, P systems provides a message-based formal framework for distributed computing, which consists of a collection of autonomous computing cells.

Chapter 1 maps distributed characteristics proposed for P systems to fundamental concepts of distributed systems, such as synchrony and asynchrony, process activity status, activation assumptions, and message and runtime complexity measures.

Chapter 2 defines xP systems, which are extended versions of simple P systems [51, 3, 52, 22, 50], with a number of ingredients that are useful or even *required* for modelling distributed algorithms, such as complex symbols, cell IDs and generic rules.

Chapters 3, 4 and 5 showcase xP systems to model asynchronous and synchronous distributed algorithms and improve large practical applications, from fundamental graph traversal algorithms and distributed termination detection problems to complex graph theoretical problems and to one of the most challenging distributed problems, the MST problem. These modelling approaches have proved the adequacy of our xP systems in modelling fundamental and challenging distributed algorithms and have provided positive answers to our main questions (§ 1.3).

- By using complex symbols and generic rules, all the work required to be done by a given complex algorithm in a logical step can be done in one single step in xP systems. Thus, our xP specifications achieve the *same* runtime complexities as the corresponding distributed algorithms.
- In terms of program size, our xP specifications are *comparable* to (or even substantially smaller than) the pseudocodes of the corresponding distributed algorithms.

- For the *synchronous* and *asynchronous* algorithms in Chapter 3, the xP system sizes are mostly half of the sizes of *distributed* pseudocodes in Tel's book [62], which show essential distributed implementation details.
- For the *synchronous* algorithms in Chapter 4, the xP system sizes are slightly larger than the sizes of our *sequentialised* pseudocodes and *structural parallel* pseudocodes, which use mostly global variables for the purpose of readability and show fewer distributed details.
- For the *synchronous* algorithm of the MST problem in Chapter 5, the xP system size is less than half of the size of our *structural parallel* pseudocodes, which use mostly local cell variables to illustrate the distributed implementation details.

The xP specifications implement every detail but the pseudocodes can show various levels of details. Typically, the pseudocodes that are higher level with fewer implementation details have smaller program size. Thus, the program size differences between our xP specifications and the pseudocodes are affected by the level of details of the pseudocodes.

Moreover, our modelling approaches also give positive answers to the following questions (§ 1.3).

- Our xP systems provide a unified model for both synchronous and asynchronous systems: the same static description for both synchronous and asynchronous systems and only the messaging delays differ. The local computation takes zero time and the messaging delay takes arbitrary time as in the distributed algorithms; this also supports theoretical fairness: a message sent is guaranteed to arrive, but there is no time bound for this. These concepts are closer to the definitions used in distributed algorithms [42, 62] and are validated by modelling synchronous and asynchronous algorithms in Chapter 3 and synchronous algorithms in Chapters 4 and 5.

- Our xP systems relate closely to the usual activation assumptions in the distributed computing framework (§ 1.1). Activation assumptions have not been addressed in P systems, where a cell can become active without receiving a message. This problem is non-trivial in distributed algorithms, e.g., in termination detection algorithms. Our λ messaging extension solves this problem: when a cell applies a rule which makes the cell active in the next step, it sends a λ message to itself. This is validated by modelling termination detection algorithms, which work based on activation assumptions.

- Our xP systems provide adequate data structures, i.e. complex symbols, for complex algorithms. Although elementary symbols seem sufficient for many theoretical studies, our preliminary practical experience shows complex symbols are needed. Complex symbols provide cell IDs that are required in distributed

algorithms [42, 62], and also provide complex data structures that are needed to solve complex problems, which are validated by modelling disjoint paths problems in Chapter 4 and one of the most challenging problems, the MST problem, in Chapter 5.

- Our xP systems enable SoC designs for complex problems that can be divided into smaller sections and each section addresses a separate concern. Our rule fragments compositions enable SoC designs: using sequential composition, cells apply rules of a system exclusively after they finish rules of the other system; using parallel composition with no interaction, two independent systems run in parallel; using parallel composition with interaction, we can cleanly add a control layer system over a system. This feature is validated by modelling algorithms, which include a preliminary local topology discovery phase, using sequential composition in Chapters 3 and 4 and by modelling termination detection algorithms using parallel composition with interaction in Chapters 3.

- By using complex symbols, our xP systems achieve a single solution with fixed-size alphabets and rulesets, where the number of rules does not depend on the number of cells in the underlying P system. This is validated by the crisp and concise specifications of all distributed algorithms in this thesis.

For future work, we intent to model other challenging distributed problems, such as modern Byzantine fault tolerance [7, 8, 36] and the asynchronous GHS algorithm [62], and investigate other distributed features, such as fairness, safety, liveness and infinite executions. We can also consider using functional programming elements allowed by complex symbols [50], such as local functions, to enable a design where rules of fundamental algorithms can be separately built as local functions to be reused for developing complex distributed algorithms.

Bibliography

[1] Aggarwal, A., Kleinberg, J., Williamson, D.P.: Node-disjoint paths on the mesh and a new trade-off in VLSI layout. SIAM J. Comput. 29(4), 1321–1333 (2000)

[2] Awerbuch, B.: A new distributed depth-first-search algorithm. Inf. Process. Lett. 20(3), 147–150 (1985)

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

[4] Bernardini, F., Gheorghe, M., Margenstern, M., Verlan, S.: How to synchronize the activity of all components of a P system? Int. J. Found. Comput. Sci. 19(5), 1183–1198 (2008)

[5] Borůvka, O.: O jistém problému minimálním (about a certain minimal problem). Práce mor. přírodověd. spol. v Brně III 3, 37–58 (1926)

[6] Casiraghi, G., Ferretti, C., Gallini, A., Mauri, G.: A membrane computing system mapped on an asynchronous, distributed computational environment. In: Freund, R., Păun, G., Rozenberg, G., Salomaa, A. (eds.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 3850, pp. 159–164. Springer (2005)

[7] Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: Proceedings of the Third Symposium on Operating Systems Design and Implementation. pp. 173–186. OSDI '99, USENIX Association, Berkeley, CA, USA (1999)

[8] Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. 20(4), 398–461 (2002)

[9] Cavaliere, M., Egecioglu, O., Ibarra, O., Ionescu, M., Păun, G., Woodworth, S.: Asynchronous spiking neural P systems: Decidability and undecidability. In: Garzon, M., Yan, H. (eds.) DNA Computing, Lecture Notes in Computer Science, vol. 4848, pp. 246–255. Springer Berlin Heidelberg (2008)

[10] Cavaliere, M., Ibarra, O.H., Păun, G., Egecioglu, O., Ionescu, M., Woodworth, S.: Asynchronous spiking neural P systems. Theor. Comput. Sci. 410, 2352–2364 (2009)

[11] Cavaliere, M., Mura, I.: Experiments on the reliability of stochastic spiking neural P systems. Natural Computing 7, 453–470 (2008)

[12] Cavaliere, M., Sburlan, D.: Time and synchronization in membrane systems. Fundam. Inf. 64, 65–77 (2004)

[13] Cidon, I.: Yet another distributed depth-first-search algorithm. Information Processing Letters 26, 301–305 (1988)

[14] Ciobanu, G., Pérez-Jiménez, M.J., Păun, G.: Applications of Membrane Computing (Natural Computing Series). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)

[15] Clocksin, W.F., Mellish, C.S.: Programming in Prolog. Springer-Verlag Berlin Heidelberg (2003)

[16] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)

[17] Dinneen, M.J., Kim, Y.B., Nicolescu, R.: Edge- and node-disjoint paths in P systems. Electronic Proceedings in Theoretical Computer Science 40, 121–141 (2010)

[18] Dinneen, M.J., Kim, Y.B., Nicolescu, R.: Edge- and vertex-disjoint paths in P modules. In: Ciobanu, G., Koutny, M. (eds.) Workshop on Membrane Computing and Biologically Inspired Process Calculi. pp. 117–136 (2010)

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

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

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

[22] ElGindy, H., Nicolescu, R., Wu, H.: Fast distributed DFS solutions for edge-disjoint paths in digraphs. In: Csuhaj-Varjú, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, G. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 7762, pp. 173–194. Springer-Verlag, Berlin Heidelberg (2013)

[23] Elkin, M.: A faster distributed protocol for constructing a minimum spanning tree. Journal of Computer and System Sciences 72(8), 1282 – 1308 (2006)

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

[25] Freund, R.: Asynchronous P systems and P systems working in the sequential mode. In: Membrane Computing, Lecture Notes in Computer Science, vol. 3365, pp. 36–62. Springer Berlin Heidelberg (2005)

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

[27] Frisco, P.: P systems, Petri nets, and program machines. In: Freund, R., Păun, G., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 3850, pp. 209–223. Springer Berlin Heidelberg (2006)

[28] Gallager, R.G., Humblet, P.A., Spira, P.M.: A distributed algorithm for minimum-weight spanning trees. ACM Trans. Program. Lang. Syst. 5(1), 66–77 (1983)

[29] Ghosh, S.: Distributed Systems: An Algorithmic Approach. Chapman & Hall/CRC Computer and Information Science Series, CRC Press (2010)

[30] Grygorash, O., Zhou, Y., Jorgensen, Z.: Minimum spanning tree based clustering algorithms. In: Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence. pp. 73–81. ICTAI '06, IEEE Computer Society, Washington, DC, USA (2006)

[31] Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J.: Depth-first search with P systems. In: Gheorghe, M., Hinze, T., Păun, G., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 6501, pp. 257–264. Springer Berlin Heidelberg (2011)

[32] Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using NetworkX. In: Proceedings of the 7th Python in Science Conference (SciPy2008). pp. 11–15. Pasadena, CA USA (2008)

[33] Ipate, F., Nicolescu, R., Niculescu, I.M., Stefan, C.: Synchronization of P Systems with Simplex Channels. ArXiv e-prints (Aug 2011)

[34] Khan, M., Pandurangan, G.: A fast distributed approximation algorithm for minimum spanning trees. In: Proceedings of the 20th International Conference on Distributed Computing. pp. 355–369. DISC'06, Springer-Verlag, Berlin, Heidelberg (2006)

[35] Kim, Y.B.: Distributed Algorithms in Membrane Systems. Ph.D. thesis, University of Auckland (2012)

[36] Kirsch, J., Amir, Y.: Paxos for system builders: an overview. In: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware. pp. 1–6. LADIS '08, ACM, New York, NY, USA (2008)

[37] Kleijn, J., Koutny, M., Rozenberg, G.: Towards a Petri net semantics for membrane systems. In: Freund, R., Păunun, G., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 3850, pp. 292–309. Springer Berlin Heidelberg (2006)

[38] Knuth, D.E., Larrabee, T., Roberts, P.M.: Mathematical Writing. Mathematical Association of America, Washington, DC, USA (1989)

[39] Kozen, D.C.: The Design and Analysis of Algorithms. Springer-Verlag, New York, NY, USA (1991)

[40] Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. In: Proceedings of the American Mathematical Society. vol. 7, pp. 48–50 (1956)

[41] Lee, Y.O., Reddy, A.N.: Constructing disjoint paths for failure recovery and multipath routing. Computer Networks 56(2), 719 – 730 (2012)

[42] Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)

[43] Makki, S.A.M., Havas, G.: Distributed algorithms for depth-first search. Inf. Process. Lett. 60, 7–12 (1996)

[44] Martín-Vide, C., Păun, G., Pazos, J., Rodríguez-Patón, A.: Tissue P systems. Theor. Comput. Sci. 296(2), 295–326 (2003)

[45] Metta, V., Krithivasan, K., Garg, D.: Representation of spiking neural P systems with anti-spikes through Petri nets. In: Suzuki, J., Nakano, T. (eds.) Bio-Inspired Models of Network, Information, and Computing Systems, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 87, pp. 651–658. Springer Berlin Heidelberg (2012)

[46] Metta, V., Krithivasan, K., Garg, D.: Modeling spiking neural P systems using timed Petri nets. In: Nature Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on. pp. 25–30 (2009)

[47] Nagy, B.: On efficient algorithms for SAT. In: Csuhaj-Varjú, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, G. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 7762, pp. 295–310. Springer Berlin Heidelberg (2013)

[48] Nicolescu, R.: Parallel and distributed algorithms in P systems. In: Gheorghe, M., Păun, G., Rozenberg, G., Salomaa, A., Verlan, S. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 7184, pp. 35–50. Springer, Berlin Heidelberg (2012)

[49] Nicolescu, R., Dinneen, M.J., Kim, Y.B.: Towards structured modelling with hyperdag P systems. International Journal of Computers, Communications and Control 2, 209–222 (2010)

[50] Nicolescu, R., Ipate, F., Wu, H.: Towards high-level P systems programming using complex objects. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Y. (eds.) 14th International Conference on Membrane Computing (CMC14), Chisinau, Republic of Moldova, August 20-23, 2013, Proceedings. pp. 255–276 (2013)

[51] Nicolescu, R., Wu, H.: BFS solution for disjoint paths in P systems. In: Calude, C., Kari, J., Petre, I., Rozenberg, G. (eds.) Unconventional Computation, Lecture Notes in Computer Science, vol. 6714, pp. 164–176. Springer, Berlin Heidelberg (2011)

[52] Nicolescu, R., Wu, H.: New solutions for disjoint paths in P systems. Natural Computing 11, 637–651 (2012)

[53] Nicolescu, R., Wu, H.: Complex objects and applications. In: Zhang, G., Wang, J., Cheng, J., Wang, T. (eds.) Proceedings of the Asian Conference on Membrane Computing (ACMC2013), November 4-7, 2013, Chengdu, China. pp. 179–198 (2013)

[54] Pan, L., Zeng, X., Zhang, X.: Time-free spiking neural P systems. Neural Computation 23, 1320–1342 (2011)

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

[56] Păun, G.: Membrane Computing: An Introduction. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2002)

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

[58] Prim, R.C.: Shortest connection networks and some generalizations. Bell System Technology Journal 36, 1389–1401 (1957)

[59] Salipante, S.J., Hall, B.G.: Inadequacies of minimum spanning trees in molecular epidemiology. Journal of Clinical Microbiology 49(10), 3568 – 3575 (2011)

[60] Seo, D., Thottethodi, M.: Disjoint-path routing: Efficient communication for streaming applications. In: IEEE International Parallel Distributed Processing Symposium. pp. 1–12. IEEE (2009)

[61] Sharma, M.B., Iyengar, S.S.: An efficient distributed depth-first-search algorithm. Inf. Process. Lett. 32, 183–186 (1989)

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

[63] Weiss, G. (ed.): Multiagent systems: a modern approach to distributed artificial intelligence. MIT Press, Cambridge, MA, USA (1999)

[64] Wu, H.: Minimum spanning tree in P systems. In: Pan, L., Păun, G., Song, T. (eds.) Proceedings of the Asian Conference on Membrane Computing (ACMC2012), Huazhong University of Science and Technology, October 15-18, 2012, Wuhan, China. pp. 88–104 (2012)

[65] Wu, H.: Fast distributed BFS solution for edge-disjoint paths. In: Yin, Z., Pan, L., Fang, X. (eds.) Proceedings of The Eighth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), 2013, Advances in Intelligent Systems and Computing, vol. 212, pp. 1003–1011. Springer Berlin Heidelberg (2013)

[66] Xu, Y., Uberbacher, E.C.: 2D image segmentation using minimum spanning trees. Image and Vision Computing 15(1), 47 – 57 (1997)

[67] Yang, L., Guo, M.: High-Performance Computing: Paradigm and Infrastructure. Wiley Series on Parallel and Distributed Computing, John Wiley & Sons (2005)

[68] Yuan, Z., Zhang, Z.: Asynchronous spiking neural P system with promoters. In: Xu, M., Zhan, Y., Cao, J., Liu, Y. (eds.) Advanced Parallel Processing Technologies, Lecture Notes in Computer Science, vol. 4847, pp. 693–702. Springer Berlin Heidelberg (2007)

[69] Zahary, A., Ayesh, A.: An analytical review for multipath routing in mobile ad hoc networks. Int. J. Ad Hoc Ubiquitous Comput. 5(2), 69–85 (2010)