On the Feasibility of MapReduce to Compute Phase Space Properties of Graphical Dynamical Systems: An Empirical Study

Tania Hamid

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Madhav V. Marathe, Chair
Chris J. Kuhlman
Calvin J. Ribbens
V. S. Anil Vullikanti

May 14, 2015
Blacksburg, Virginia

Keywords: Graph Dynamical Systems, GDS, MapReduce, Map, Reduce, Hadoop

On the Feasibility of MapReduce to Compute Phase Space Properties of Graphical Dynamical Systems: An Empirical Study

Tania Hamid

(ABSTRACT)

A graph dynamical system (GDS) is a theoretical construct that can be used to simulate and analyze the dynamics of contagion processes on network representations of systems. One of our goals is to compute the phase space of a system (i.e., the set of all transitions for each state of the system), and for this, even 30-vertex graphs present a computational challenge. This is because the number of states is exponential in the number of graph vertices. We implement several algorithms for phase space computations based on naive approaches, and based on more recent graph dynamical systems theory. We find that MapReduce is not well suited for these types of computations. While we observe improvements in execution times with algorithmic improvements, these improvements are not sufficient to overcome inefficiencies of MapReduce for these types of computations. We present a detailed set of observations explaining system performance. Future work entails executing our algorithms on Apache Spark.

Acknowledgments

First and foremost, I thank my advisor, Dr. Madhav V. Marathe, for giving me the wonderful opportunity to work with him and to be a part of Network Dynamics and Simulation Science Laboratory. He has always inspired me with his immense knowledge and foresightedness. I feel privileged to have him as my advisor.

I offer my heartfelt gratitude to my mentor and co-advisor, Dr. Chris J. Kuhlman, for working with me. This thesis would not have been possible without his invaluable insights, constant support, guidance and motivation. In spite of his busy schedule, he has always found time for me and given me his reviews and feedback. I thank him for his patience and for his trust in my capabilities.

I would like to thank Dr. V. S. Anil Vullikanti and Dr. Calvin J. Ribbens for supporting me by being on my committee.

By working at NDSSL, I have got the opportunity to do a lot of quality research on diverse areas. I thank Dr. Anil Vullikanti, Dr. Achla Marathe, all my colleagues and friends at NDSSL for this enriching experience.

I am grateful to my friends and family for their constant love, support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Many systems (human and otherwise) can be represented as networks, where vertices represent the actors of a system and edges represent their interactions. These systems span diverse domains, such as telecommunication systems, power systems, sociology and biology. These systems evolve in response to within-system interactions and exogenous influences. Understanding the processes and mechanisms involved in these is a big challenge. Simulations are used to model networked systems and study their system dynamics. These analyses are of practical interest since they provide insights into the behaviors of such systems and how to control them.

Graph dynamical systems (GDSs) are used as a mathematical framework to model and simulate various phenomena occurring in complex networks. They can be used to study a wide array of dynamical processes, including social unrest in human population, disease propagation on social contact graphs, packet flow in cell phone communication, and urban traffic and transportation.

Informally, a GDS consists of a dependency graph; a set of vertex states, an element of which is assigned to each vertex; a function for each vertex that describes how the vertex changes its state and an update mechanism that specifies the sequence in which vertex functions are executed. Vertex functions have dependencies which are encoded by the dependency graph of the GDS. The dynamics of a networked system, that is modeled as a GDS, is computed by evaluating vertex functions on each vertex of the GDS at each time step. A formal definition for GDS can be found in Chapter 2. GDS is a useful construct to study the **phase space** of a networked system i.e. the set of all system state transitions for all system states. A serial software application, modeling a GDS, has been used in three works [47, 45, 46] to experimentally identify dynamical system behaviors which have been then rigorously proved as general characterizations of GDSs.

Computing the complete dynamics of a GDS can require calculation of $n! \cdot 2^n$ system state transitions for a Boolean GDS with an n-vertex graph. Figure 1.1 shows that the number of state transitions to be computed varies exponentially with the size of the graph. Thus, as networked systems grow in size, computation of their dynamics becomes more expensive or practically impossible. This poses a big challenge in GDS analysis, specifically on serial platforms. We, thus, need to design robust and scalable tools that will allow us to model large complex graph dynamical systems and evaluate them at a faster speed. Such a tool should also be flexible enough to accomodate different properties of various GDSs.

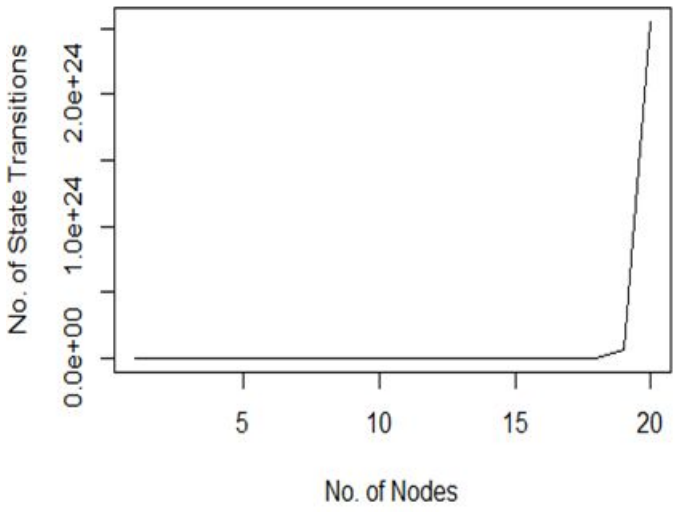


Figure 1.1: Plot showing increase in computation of number of system state transitions with size of the graph.

We have designed a GDS Evaluator tool that uses various serial and parallel programming algorithms to explore the entire phase space of a GDS. We use different theoretical concepts of dynamical systems and study how these concepts allow us to obtain the dynamics of a GDS, without having to compute all possible system state transitions. We use Hadoop MapReduce framework to implement all parallel GDS evaluation algorithms.

1.2 Contributions and Suitability of MapReduce for Dynamical Systems Computations

First, we provide our contributions and then discuss the suitability of MapReduce (Hadoop implementation) for the dynamical systems computations of this study.

Our contributions are:

1. **Design, implementation and evaluation of different programming paradigms to perform phase space computations.** We design various serial and parallel programming modules for performing phase space computations. We implement the serial paradigms in C++ and the parallel paradigms on the Hadoop MapReduce framework using the Java MapReduce API. We analyze the time and space complexities of each of these algorithms and study their suitability in modeling GDSs. We also perform different experiments on various GDSs to evaluate the performance of these approaches. We find that the serial code performs just as well as the MapReduce implementations for smaller networks, and memory issues prevent efficient execution on larger networks.

2. **Quantitative demonstration that the Hadoop MapReduce implementation is not well suited in its current form for network simulations of the kind described herein.** Implementations of various GDS computations demonstrate that the Hadoop MapReduce framework is not suitable in its current form for network simulations. We provide performance data comparisons between serial implementations and MapReduce-based implementations of GDS computations, and show that in many cases, the serial code out-performs MapReduce. We also explore multiple algorithms, using more advanced dynamical systems theory, that are implemented with Hadoop, and while these improve performance, the magnitude of this improvement is not sufficient to make the Hadoop MapReduce attractive. This is elaborated upon below.

3. **Use GDS theory to devise algorithms to perform phase space computations for sequential GDSs.** We use different theoretical concepts of dynamical systems to design algorithms to compute phase spaces of a sequential GDS. These algorithms use different vertex function agnostic (VFA) techniques. VFA techniques enable us to compute various dynamical characteristics of a GDS, without having to compute any system state transitions. We study how VFA techniques based on update graphs and acyclic orientations of sequential update sequences can be used to perform VFA GDS evaluation. We demonstrate how application of theory can facilitate more efficient phase space computations.

4. **Addition of new features to the GDS Evaluator tool.** These features enable us to model GDSs having multi-dimensional vertex states and to perform phase space computation on reduced vertex state dimensions. These new features also allow for rule-based vetex function assignment and support the ability to choose open or closed neighborhoods of a vertex to perform phase space computations. These features are discussed in detail in Chapter 6.

MapReduce has shortcomings for performing some types of computations; e.g., iterative computations [22]. However, Lin [52] argues that MapReduce may be "good enough" for many types of problems, including iterative problems similar to the one we investigate here. We undertake this study to determine whether MapReduce is a viable candidate for our type of iterative computation. We use both iterative and non-iterative MapReduce algorithms to

perform different GDS computations and compare their performances. In addition, we study the effects of algorithmic improvements on performance. We compare the performance of brute-force MapReduce implementations of GDS computations to that of implementations based on more advanced dynamical systems theory. In many domains, large improvements in software performance come with algorithmic improvements [5, 6]. Here, while these more advanced implementations do improve performance, the improvements are not enough to make the overall MapReduce approach viable. Our work, like other works [22, 38], is a quantitative study of the shortcomings of MapReduce framework for the computations we describe herein.

We now discuss issues contrasting iterative and non-iterative approaches that we use to perform GDS computations on the Hadoop MapReduce framework. We use four different MapReduce-based approaches to perform GDS computations. These are the non-iterative brute force GDS evaluation paradigm, the iterative brute force GDS evaluation paradigm, the non-iterative VFA GDS evaluation paradigm and the iterative VFA GDS evaluation paradigm. Chapter 8 discusses these in detail. The iterative brute force GDS evaluation paradigm and the iterative VFA GDS evaluation paradigm use iterative MapReduce jobs; while the non-iterative brute force GDS evaluation paradigm and the non-iterative VFA GDS evalaution paradigm use non-iterative MapReduce jobs to perform different computations. For the systems that we study, we find the following:

1. **The iterative brute force approach takes the longest time to perform GDS computations**. Each iteration corresponds to a separate MapReduce job, which has an associated job latency [67]. This latency incurs significant performance overhead for large number of iterations. Moreover, each iteration involves reading GDS configuration parameters from files residing on the HDFS, reading and writing key-value pairs from and to files on the HDFS and shuffling and sorting intermediate key-value pairs. Though the values for GDS configuration parameters are constant over successive iterations, they are read in each iteration as Hadoop is incapable of caching data across multiple MapReduce jobs. All these add significant overhead. The iterative brute force approach runs for a large number of iterations. For example, the number of iterations for a sequential Boolean system consisting of a 8-node Circle graph is 256. Also, the number of iterations increase exponentially with the increase in the number of nodes in the dependency graph of a GDS. As a result, GDS evaluation using the iterative brute force approach has high performance overheads, which result in an increase in its execution time.

2. **The iterative VFA approach performs better compared to the iterative brute force approach**. Algorithmic improvements cause the iterative VFA approach to require fewer number of iterations compared to the iterative brute force approach.

3. **The non-iterative approaches perform better than the iterative approaches when the size of a GDS is small.** This is because for small networks, the overhead associated with non-iterative approaches is smaller than that with iterative approaches.

4. **The iterative VFA approach performs better than the non-iterative approaches with increase in the size of the dependency graph of a GDS.** This is because the map and reduce phases of the MapReduce jobs used in non-iterative approaches process larger volumes of key-value pairs compared to the total number of key-value pairs processed by all the iterations of the iterative VFA approach. For example, the number of key-value pairs processed by all the jobs used in the non-iterative brute force approach for a 8-node Circle graph is 6.5×10^7; while that processed by all jobs used by the iterative VFA approach is 2.6×10^6. This causes the communication cost [4] of these jobs to be very high compared to the cost of running the iterative MapReduce jobs.

The Hadoop MapReduce framework is unsuitable for GDS computations due to the following reasons. These issues are valid irrespective of the MapReduce approach that we use to perform GDS computations.

1. **GDS evaluation using Hadoop MapReduce is memory intensive.** This is due to the inherent nature of the Hadoop MapReduce framework and due to the nature of the key-value pairs produced by various MapReduce jobs used in GDS computations. Hadoop MapReduce framework sorts all intermediate key-value pairs generated by the map phase of a MapReduce job. These key-value pairs are sorted by their keys. Since MapReduce jobs used for GDS computations produce large volumes of long intermediate key-value pairs, this sorting is costly [39] and is memory intensive.

2. **GDS evaluation using MapReduce is I/O intensive.** A Hadoop MapReduce job performs a series of disk operations to generate its output. It reads all its input key-value pairs from an input file residing on HDFS, the distributed file system for Hadoop. All intermediate key-value pairs produced by the map phase of a MapReduce job are written to temporary files. All key-value pairs produced by the reduce phase of a MapReduce job are written to the HDFS. Thus each MapReduce job needs to access the disk multiple times, which is costly. As discussed in subsequent chapters, we perform different GDS computations by chaining multiple Hadoop MapReduce jobs. This further increases the disk access and introduces performance overheads. Some improvement is expected with the use of Hadoop Streaming API.

3. **GDS evaluation using MapReduce is network intensive.** This is because MapReduce jobs performing GDS computations produce a large number of intermediate key-value pairs. Also, the length of these keys increases exponentially with the size of the GDS. This results in shuffling of large volumes of data over the network between map and reduce tasks that run on different compute nodes of the Hadoop cluster.

Now we discuss some generic issues which affect the performance of any MapReduce job running on the Hadoop framework. They are:

1. **Memory allocations are hardware specific.** Thus any alteration that is made for one user's memory needs must be used by all system users.

2. **It is difficult to tune Hadoop without administrative access to configuration files.** Most performance tuning parameters of Hadoop cannot be overridden by a user using command line options. The user can override the default values of such parameters by editing the corresponding configuration files only. This requires the user to have administrative access to such files.

3. **Tuning the number of mapper tasks is difficult.** A user cannot explicitly specify the number of mapper tasks that he/she wants to use for their computations. Hadoop determines this at runtime depending on the size of the input file to a MapReduce job and the minimum and maximum size of each chunk that this file should be split into. A user can override the default values for the latter, but this has to be done for every MapReduce job explicitly.

These shortcomings make the Hadoop MapReduce framework less suitable for performing GDS computations. Nevertheless, we can use this framework to study small networks. We can study bigger GDSs using Hadoop if we have larger clusters at our disposal. Alternate systems such as Apache Giraph, Apache Spark, etc., are better suited to model GDS-like systems. All MapReduce algorithms that we have used in our work can be ported and re-implemented on Apache Spark using its APIs.

Chapter 2

Graph Dynamical Systems

This chapter provides a formal description of the GDS model. We define different elements that constitute a GDS. We also discuss different techniques that can be used to evaluate a GDS. Variants of this discussion can be found in [47, 44, 55, 49]. Besides being useful for software implementation, GDS theory is also useful for exploring analysis problems [8, 9, 11, 10].

2.1 GDS Formalism

A GDS is denoted by $\mathcal{S}(X, F, \mathcal{W}, K)$. Let X denote a directed graph, called a **dependency graph**, with vertex set $\mathrm{v}[X] = \{1, 2, \ldots, n\}$ and edge set $\mathrm{e}[X]$. We use the convention that a directed edge (u, v) implies that the state of vertex u is used to determine the next state of vertex v. Often the convention is adopted that self loops (i.e. the edge (v, v)) are not drawn on the dependency graph in order to reduce clutter. Throughout this work, we will not draw self loops and will assume that $v \in n[v]$ for all $v \in v[X]$. Each vertex v is assigned a state $x_v \in K$ and is referred to as the **vertex state**. K is the **vertex state set** and is given by $K = \{0, 1, \ldots, r-1\}$, for some fixed integer $r \geq 2$. The state of all vertices in the dependency graph constitutes the **system state** and is given by $x = (x_1, x_2, \ldots, x_n)$. The 1-neighborhood of a vertex v is the set of vertices adjacent to v in X. Let $n[v]$ denote the sequence of vertices in the 1-neighborhood of vertex v sorted in increasing order such that for each $u \in n[v]$, there exists a directed edge $(u, v) \in \mathrm{e}[X]$. (The 1-neighborhood is closed if $v \in n[v]$, implying that there is a directed self-loop.) In other words, each such u is an in-neighbor of v, and $d^{in}(v) = |n[v]|$, where d^{in} is the in-degree of v. We write the sequence $x[v]$ of vertex states corresponding to the vertices in $n[v]$ as

$$x[v] = \left(x_{n[v](1)}, x_{n[v](2)}, \ldots, x_{n[v](d^{in}(v))}\right) .$$

We refer to $x[v]$ as the **restricted state**. Here, $n[v](i)$ is the ith entry in the sequence. We denote the (system) state and restricted state at time t as $x(t)$ and $x(t)[v]$, respectively.

The dynamics of changes in vertex states are governed by a sequence $F = (f_v)_{v=1}^n$ of **vertex functions** where each $f_v \colon K^{d^{in}(v)} \longrightarrow K$ maps as

$$x_v(t+1) = f_v\big(x(t)[v]\big) \ .$$

In other words, the state of vertex v at time $t+1$ is obtained by evaluating f_v on the restricted state $x[v]$ at time t. To reduce notation, we will often omit the time t from the restricted state.

An **update scheme** $\mathcal{W}$ governs how the list of vertex functions assemble to a **graph dynamical system map** (see e.g. [58, 54])

$$\mathbf{F} \colon K^n \longrightarrow K^n$$

producing the system state at time $t+1$ from that at time t; i.e., $x(t+1) = \mathbf{F}(x(t))$. Hence the GDS map governs how the system state evolves over time, through vertex state transitions. We consider three different types of update mechanisms: **synchronous update**, **sequential update** and **block sequential update**. Each of these update schemes is explained as follows:

- **Synchronous Update:** The synchronous (parallel) GDS map for this scheme is given by
$$F(x_1, \ldots, x_n) = (f_1(x[1]), f_2(x[2]), \ldots, f_v(x[v]), \ldots, f_n(x[n]))$$
We refer to this subclass of GDS as **synchronous dynamics systems** (SyDS), since all vertex functions are executed simultaneously (i.e., in parallel). It is sometimes also referred to as **generalized cellular automata**.

- **Sequential Update or Asynchronous update:** This scheme uses permutation update sequences and can be explained using the concept of X**-local functions**. The X-local function $F_v \colon K^n \longrightarrow K^n$ is given by
$$F_v(x_1, \ldots, x_n) = (x_1, \ldots, x_{v-1}, f_v(x[v]), x_{v+1}, \ldots, x_n) \ ;$$
i.e., F_v updates only the vth component of the system state. Using $\pi = (\pi_1, \ldots, \pi_n) \in S_X$ (the set of all permutations of v$[X]$) as an update sequence, the corresponding asynchronous (or sequential) GDS map $\mathbf{F}_\pi \colon K^n \longrightarrow K^n$ is given by
$$\mathbf{F}_\pi = F_{\pi_n} \circ F_{\pi_{n-1}} \circ \cdots \circ F_{\pi_2} \circ F_{\pi_1} \ ,$$
which is the composition of the X-local functions. We refer to this class of asynchronous systems as **(permutation) sequential dynamical systems** (SDS).

- **Block Sequential Update:** Block sequential update mechanism is a generalization of the previous two schemes. In this scheme, the vertices are partitioned into a sequence of q sets or blocks $B = (B_1, B_2, \ldots, B_q)$. The vertex functions for the vertices in

each block are executed simultaneously, with sequential ordering between consecutive blocks. Let $\pi_B = (\pi_{B_1}, \ldots, \pi_{B_q})$ be a block permutation. We have the X-local function, for $l \in \{1, \ldots, q\}$, $F_{\pi_{B_l}} : K^n \longrightarrow K^n$, defined by

$$\left(F_{\pi_{B_l}}(x)\right)_v = \begin{cases} f(x[v]), & \text{if } v \in B_l, \\ x_v, & \text{otherwise.} \end{cases} \tag{2.1}$$

We have the block sequential GDS map

$$\mathbf{F}_{\pi_B} = F_{\pi_{B_q}} \circ F_{\pi_{B_{q-1}}} \circ \cdots \circ F_{\pi_{B_2}} \circ F_{\pi_{B_1}} \ ,$$

and refer to it as a **block sequential dynamical system** (BSDS). When the size of each block is one, a block sequential GDS map reduces to a sequential GDS map, and when all vertices are in one block, the block sequential map reduces to a synchronous GDS map. To this point, we have described **fair word orders**; that is, each vertex appears exactly once in a (block) permutation. **Unfair word orders** [58], where vertices may appear more than once in a (block) permutation, are also studied. If unspecified, the convention is to assume a fair word order.

We describe all three types of maps here because different works in the literature may use only one of the update methods. Our work, however, focuses on sequential and synchronous update mechanisms only.

2.2 Phase Space

The **phase space** $\Gamma(\mathbf{F})$ of the GDS map $\mathbf{F}$ is represented using a directed graph with vertex set K^n and edge set $\{\big(x, \mathbf{F}(x)\big) \mid x \in K^n\}$. Hence the phase space represents all possible system state transitions for a given GDS map/update sequence. A state x for which there exists a positive integer p such that $\mathbf{F}^p(x) = x$ is a **periodic point**, and the smallest such integer p is the **period** of x. If $p = 1$ we call x a **fixed point** of $\mathbf{F}$. A state that is not periodic is a **transient state**. Classically, the **omega-limit set** of x, denoted by $\omega(x)$, is the set of accumulation points of the sequence $\{\mathbf{F}^k(x)\}_{k \geq 0}$. In the finite case, the omega-limit set (also called a **(limit) cycle**, **(periodic) orbit**, **limit set**, or **attractor**) is the unique periodic orbit reached from x under $\mathbf{F}$.

Phase Space Analysis of a graph dynamical system is done by computing all phase spaces corresponding to all possible GDS maps of an update sequence. Thus, a sequential dynamical system consisting of an n node graph will yield $n!$ phase spaces corresponding to each of the $n!$ permutation of vertices. On the other hand, a synchronous dynamical system will contain only one phase space in its phase space analysis.

The phase space of a GDS can be analyzed to obtain the following:

- **Functional Equivalence Class**
  If two GDS maps $\mathbf{F}_1$ and $\mathbf{F}_2$ give the exact same state transitions; i.e., $\mathbf{F}_1(x) = \mathbf{F}_2(x)$ for all $x \in K^n$, then the GDS maps are equal; i.e., $\mathbf{F}_1 = \mathbf{F}_2$, and we say that the maps are **functionally equivalent**.

- **Dynamically Equivalence Class**
  Two GDS maps $\mathbf{F}_1$ and $\mathbf{F}_2$ are dynamically equivalent if their sets of state transitions are isomorphic. That is, $\Gamma(\mathbf{F}_1)$ and $\Gamma(\mathbf{F}_2)$ are isomorphic as directed graphs. More formally, if there exists a bijection $\phi\colon K^n \to K^n$ such that $\mathbf{F}_1 \circ \phi = \phi \circ \mathbf{F}_2$, then $\mathbf{F}_1$ and $\mathbf{F}_2$ are **dynamically equivalent**.

- **Cycle Equivalence Class**
  If two GDS maps produce the same limit cycle structures (cycle lengths and multiplicities), to within an isomorphism, then we say the two maps are cycle equivalent [54]. Formally, let $\mathrm{Per}(\mathbf{F}_1)$ and $\mathrm{Per}(\mathbf{F}_2)$ be the sets of limit sets or periodic orbits of GDS maps $\mathbf{F}_1$ and $\mathbf{F}_2$. If there is a bijection $h\colon \mathrm{Per}(\mathbf{F}_1) \to \mathrm{Per}(\mathbf{F}_2)$ such that $\mathbf{F}_1\Big|_{Per(\mathbf{F}_1)} \circ h = h \circ \mathbf{F}_2\Big|_{Per(\mathbf{F}_2)}$, then $\mathbf{F}_1$ and $\mathbf{F}_2$ are **cycle equivalent**.

If two GDS maps are functionally equivalent, then they are dynamically equivalent. If two GDS maps are dynamically equivalent, then they are cycle equivalent. Functional, dynamic, and cycle equivalences can be computed for any pair of GDSs. Examples addressing these equivalence classes are given in Section 2.4.

2.3 Example Vertex Functions

First, we introduce threshold and bithreshold vertex functions. In this example, we confine ourselves to Boolean systems so that $K = \{0, 1\}$. We write d^{in} to denote $d^{in}(v)$ and assume that $v \in n[v]$. A Boolean **threshold function** $\theta_{k,d^{in}}\colon K^{d^{in}} \longrightarrow K$ is defined as

$$\theta_{k,d^{in}}(x_1, \ldots, x_{d^{in}}) = \begin{cases} 1, & \text{if } \sigma(x_1, \ldots, x_{d^{in}}) \geq k \quad \text{and} \\ 0, & \text{otherwise,} \end{cases} \tag{2.2}$$

where $\sigma(x_1, \ldots, x_{d^{in}}) = \big|\{1 \leq j \leq d^{in} \mid x_j = 1\}\big|$.

Threshold functions are used in modeling biological systems [40, 20], and social behaviors (e.g., joining a revolt, technology adoption, spread of rumors, and other social contagions), see, e.g., [32, 56, 14, 76]. A **bi-threshold function** is a function $\theta_{v,k_{01},k_{10},d^{in}}\colon K^{d^{in}} \longrightarrow K$ defined by

$$\theta_{v,k_{01},k_{10},d^{in}}(x_1, \ldots, x_{d^{in}}) = \begin{cases} \theta_{k_{01},d^{in}}, & \text{if } x_v = 0, \\ \theta_{k_{10},d^{in}}, & \text{if } x_v = 1\ . \end{cases} \tag{2.3}$$

We call k_{01} the **up-threshold** and k_{10} the **down-threshold**. When $k_{01} = k_{10}$ the bi-threshold function behaves like a standard threshold function. The up-threshold k_{01} denotes the minimum number of vertices in $n[v]$ that are required to be in state 1 in order for v to transition to 1 when its state is 0. When $x_v = 1$, if the number of vertices in $n[v]$ that are in state 1 (including v) is $< k_{10}$, then v changes to state 0. Otherwise x_v does not change.

A second vertex function is the nor function, nor: $K^{d^{in}} \longrightarrow K$, defined by

$$\text{nor}(x_1, \ldots, x_{d^{in}}) = \Pi_{j=1}^{d^{in}}(1 + x_j) \tag{2.4}$$

where all $(1 + x_j)$ are modulo 2. Hence, the only way for a nor vertex function to evaluate to 1 is for all inputs to have state 0. GDSs that utilize nor functions are studied because they have interesting properties, such as limit cycles in phase spaces are not fixed points [58].

2.4 GDS: Illustrative Examples

We provide phase spaces for GDS maps where the dependency graph X is a bidirected Circle_4 graph on four vertices, the vertex state set $K = \{0, 1\}$, and all vertex functions are nor functions. We compare the phase spaces of the synchronous GDS, and particular sequential and block sequential GDSs. Figure 2.1 provides the graph and three phase spaces. The top phase space is for the synchronous GDS; the middle phase space is for a sequential GDS where $\pi = (1, 2, 3, 4)$; and the lower phase space is for a block sequential GDS with $\pi_B = ([1, 2], 3, 4)$. The block sequential permutation has blocks $B_1 = \{1, 2\}$, $B_2 = \{3\}$, and $B_3 = \{4\}$, implying that f_1 and f_2 execute in parallel, followed by f_3, and then f_4, and hence is close to the sequential permutation. Given the state (0,0,0,0), the next state is $\mathbf{F}(0, 0, 0, 0) = (1, 1, 1, 1)$ for synchronous update, $\mathbf{F}_\pi(0, 0, 0, 0) = (1, 0, 1, 0)$ for sequential update (with the particular permutation), and $\mathbf{F}_{\pi_B}(0, 0, 0, 0) = (1, 1, 0, 0)$ for the specified block permutation. Hence, the next states are different for the three GDSs.

Overall, it is apparent that the three phase spaces are different. The long-term dynamics are also different, as described by the limit cycles. The sequential GDS has one 7-cycle; the synchronous GDS contains 2-cycles with multiplicity 3 (i.e., there are three 2-cycles); and the block sequential GDS has one 2-cycle (i.e., multiplicity 1).

In each GDS, state (0,0,1,1) is a transient state (it is not an element of a limit cycle). In the sequential and synchronous systems, it is part of a transient of length 1; i.e., there is one transition from state (0,0,1,1) before the system reaches a state on a limit cycle. This is the maximum transient length for these two GDSs. In contrast, for block sequential update, state (0,0,1,1) is part of a transient of length 3. In this system, this is the maximum transient length, and there are 10 transients of length 3.

A **basin of attraction**, as used here, is the set of all states that (eventually) transitions to, or is contained in, an attractor. For example, in the synchronous GDS, the two states

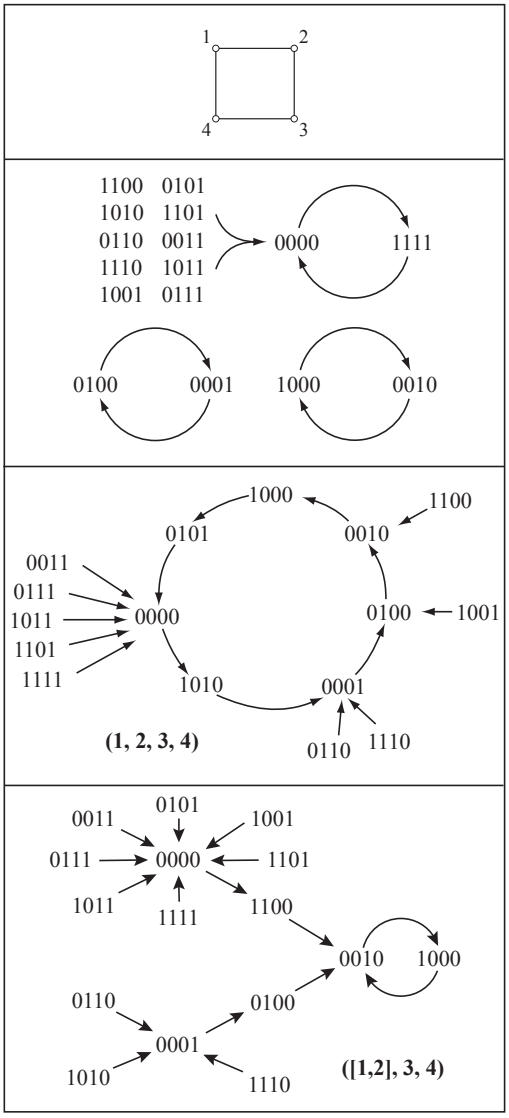


Figure 2.1: **Phase spaces for three GDSs.** The bidirected graph $X = \text{Circle}_4$ (top), and the phase spaces of the synchronous GDS; a sequential GDS with update permutation $\pi = (1, 2, 3, 4)$; and a block sequential GDS with block permutation $\pi_B = ([1, 2], 3, 4)$. All vertices use the nor function.

(0,1,0,0) and (0,0,0,1) form a 2-cycle attractor, and since there are no transient states that (eventually) transition to these two states, these two states form the basin of attraction for this attractor. There are three basins of attraction for the synchronous GDS. For each of the sequential GDS and the block sequential GDS, there is one attractor and one basin of attraction; this means that all states eventually transition to the respective limit cycle.

No pair of these three GDSs are functionally equivalent because they do not produce the same state transitions; i.e., the same phase spaces. Since the cycle lengths and the multiplicities for each cycle length are different across the three GDS maps, no two GDS maps are cycle equivalent.

As a second example, we evaluate two GDS maps that are not functionally equivalent nor dynamically equivalent, but they are cycle equivalent. Figure 2.2 provides two phase spaces for a Circle_4 graph where all vertices implement the nor function. We show GDS maps for two permutations. These GDS are cycle equivalent because the limit cycles in each consist of two 2-cycles and one 3-cycle. But they are not functionally equivalent. This is because for permutation (1,3,2,4), state (1,1,0,1) gets mapped to (0,0,0,0), and for permutation (1,4,2,3), (1,1,0,1) gets mapped to a different state, (0,0,1,0). (To show that two GDS maps are not functionally equivalent, it is enough to show that one state gets mapped to different states in the two GDS.)

The two systems are not dynamically equivalent because for permutation (1,3,2,4), the 3-cycle contains one state (state (0,0,0,0)) with maximum in-degree of 4, while for permutation (1,4,2,3), the 3-cycle has one state (state (0,0,1,0)) with maximum in-degree of 2. (This is sufficient because for dynamic equivalence, the two phase spaces must be isomorphic.)

2.5 Techniques to Compute FEECs

We compute FEECs using either a brute force approach, a vertex function agnostic approach, or both. This section formalizes these approaches and discusses the algorithms that we have used to implement them.

2.5.1 Brute Force Approach

Brute force approach computes FEECs by generating all possible system state transitions for all possible permutations of vertices of a GDS. Permutations π and π' that yield the exact same phase spaces $\Gamma(F_\pi)$ and $\Gamma(F_{\pi'})$ are grouped together into the same FEEC.

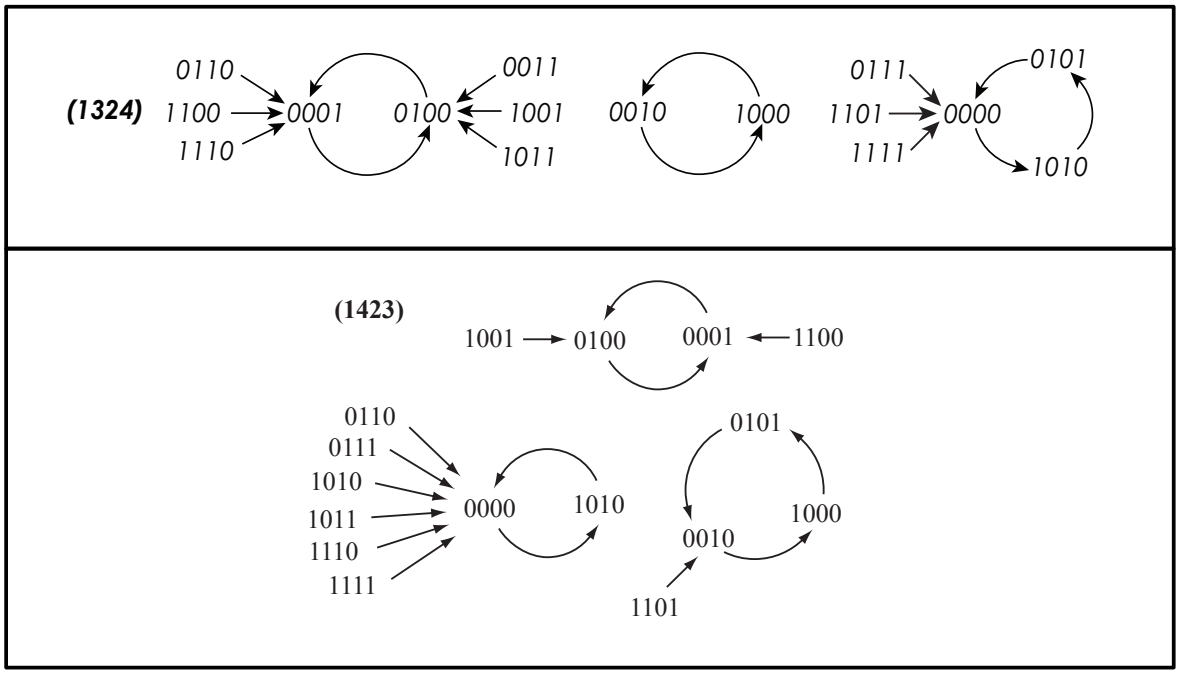


Figure 2.2: Phase spaces of two GDSs that are cycle equivalent, but not functionally equivalent or dynamically equivalent. The top phase space is that for a sequential GDS with permutation (1,3,2,4). The bottom phase space is that for a sequential GDS with permutation (1,4,2,3). All vertices use the nor function.

2.5.2 Vertex Function Agnostic Approaches

Vertex function agnostic (VFA) approaches, as used herein, are methods for computing equivalence classes of permutations. Each class of permutations produces a different phase space. The key here is that the approaches use only the permutations of vertices and the dependency graph. In particular, they do not use vertex functions nor the vertex state set. Hence, the results are applicable to any assignment of functions to vertices.

2.5.3 Permutation Update Graph

This section, and Section 2.5.4, are applicable to GDSs that use sequential update. Here we give more intuitive explanations of the models; see [54] for formal arguments.

A **permutation update graph**, or **update graph**, $U(X)$ of the dependency graph X has as vertices the permutations $\pi \in S_X$, where S_X is the set of all vertex permutations. The elements $\{\pi, \pi'\}$ of the set of undirected edges of $U(X)$ denote two permutations such that π and π' differ precisely and only in the order of exactly two consecutive elements π_i and π_{i+1} of π, where (i) $\pi'_i = \pi_{i+1}$ and $\pi'_{i+1} = \pi_i$, and (ii) π_i and π_{i+1} do not form an edge in X.

Let us take an example where $X = \text{Circle}_4$, with vertices $\text{v}[X] = \{1, 2, 3, 4\}$, as seen in Figure 2.3.

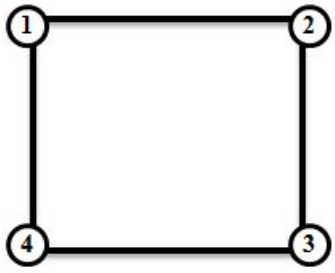


Figure 2.3: Circle graph with 4 vertices.

Consider the permutations $\pi = \{1, 2, 3, 4\}$ and $\pi' = \{1, 2, 4, 3\}$, both of which are vertices in $U(X)$. They differ in precisely two elements, π_3 and π_4, and importantly, $\pi_3 = \pi'_4$ and $\pi_4 = \pi'_3$.. However, $\{3, 4\}$ is an edge in X, and hence there is no edge $\{\pi, \pi'\}$ in the edge set of $U(X)$.

Now consider another example, where $\pi = \{1, 3, 2, 4\}$ and $\pi' = \{3, 1, 2, 4\}$. They differ in precisely two elements, π_1 and π_2. Furthermore, $\{1, 3\}$ is not an edge in X, and hence there is an edge $\{\pi, \pi'\}$ in the edge set of $U(X)$. By the same reasoning, $\pi = \{1, 2, 4, 3\}$ and $\pi' = \{1, 4, 2, 3\}$ form an edge in $U(X)$.

We appeal to intuition to understand the relevance of the update graph. Take the last

example, where $\pi = \{1, 2, 4, 3\}$ and $\pi' = \{1, 4, 2, 3\}$ form an edge in $U(X)$. Vertex 2 comes before vertex 4 in π, and the order is flipped in π'. Otherwise, the two permutations are the same. When the vertex function f_2 is executed, its arguments are $x[2] = (x_1, x_2, x_3)$. That is, x_4 is not an argument to f_2. Similarly, x_2 is not an argument for f_4 because the restricted state is $x[4] = \{x_1, x_3, x_4\}$. Thus, x_2 and x_4 do not depend on each other (i.e., they are not distance-1 neighbors in the dependency graph Circle_4). Hence, f_2 and f_4 may be executed in either order; the values of f_2 and f_4 will be the same, irrespective of order. It is not difficult to see, therefore, that for a given system state x, the next state will be x', irrespective of whether f_2 or f_4 executes before the other. Thus, it follows that $\mathbf{F}_\pi$ and $\mathbf{F}_{\pi'}$ produce the same state transitions, and are therefore functionally equivalent.

Finally, a connected component in $U(X)$ consists of a subgraph such that every vertex π can be reached from every other vertex π' in the component. By using the same arguments above, inductively, we see that all permutations of a connected component of $U(X)$ produce the same state transitions, and hence are functionally equivalent.

By implementing an algorithm that evaluates pairs of permutations that differ in precisely the same two consecutive elements, as described above, we form equivalence classes of permutations that are functionally equivalent.

In practice, one has to evaluate (some) state transitions for one permutation of each equivalence class to determine whether these permutations give the same GDS map. It is possible, owing to the particulars of the vertex functions, that two permutations in different equivalence classes may give the same state transitions, and if so, then these equivalence classes are merged to form one class, because they produce the same phase space. Nonetheless, this approach is powerful because it is vertex function agnostic, and hence one set of computations can be used as the starting point to evaluate any number of vertex function assignments.

2.5.4 Acyclic Orientations

The concept of **acyclic orientations** [54] is directly related to the ideas of the update graph, which will be described in this section. An **orientation** of an undirected graph X is a mapping from an undirected edge to a directed edge; i.e., $O(X)\colon \text{e}[X] \longrightarrow \text{v}[X] \times \text{v}[X]$. An acyclic orientation is an orientation $O(X)$ such that the graph of O has no cycles.

We define the mapping from an undirected edge to a directed edge in the following fashion. An undirected edge $\{u, v\}$ maps to (u, v); i.e., the directed edge from tail vertex u to head vertex v, if u comes before v in π. Otherwise, $\{u, v\}$ maps to (v, u). This mapping is guaranteed to be acyclic because the first element π_1 in a permutation π is always the tail vertex of all directed edges $(\pi_1, \cdot)$ involving π_1, and likewise, the last element π_n in π is always the head vertex of all directed edges $(\cdot, \pi_n)$ involving π_n.

As an example (see Figure 2.4), consider $X = \text{Circle}_4$, and let $\pi = (2, 4, 1, 3)$. The acyclic orientation $O(X)$ gives the set of directed edges $\{(2, 1), (2, 3), (4, 3), (4, 1)\}$. The first entry

in the set, $(2, 1)$, is the directed edge from vertex 2 to vertex 1 because 2 comes before 1 in π. There are, in general, multiple permutations that will generate the same set of directed

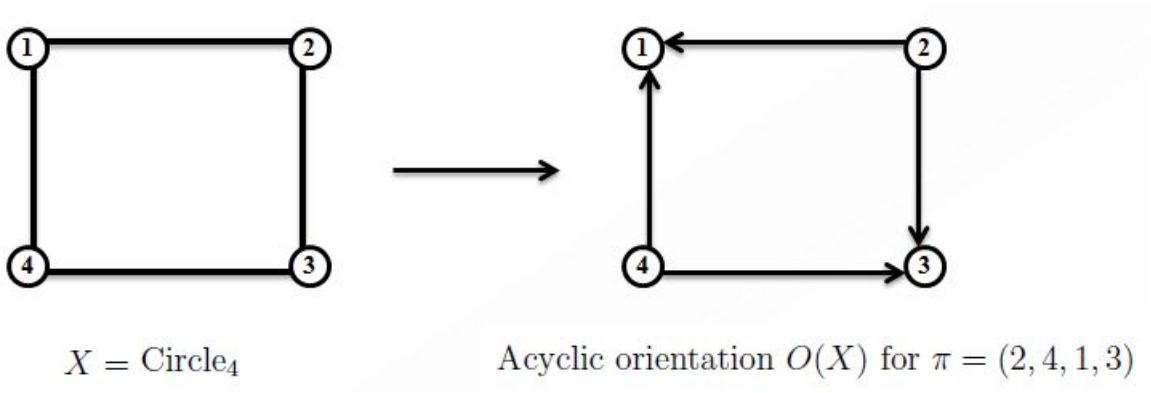


Figure 2.4: Acyclic orientation $O(X)$ for X=Circle$_4$ and $\pi = (2, 4, 1, 3)$.

edges. For example, the permutation $(4, 2, 3, 1)$ will produce the same set of directed edges as $(2, 4, 1, 3)$; i.e., it will produce the same acyclic orientation. The permutation $(1, 4, 3, 2)$ will generate a different acyclic orientation. The set of all acyclic orientations of X is designated Acyc(X).

The goal is to assign an acyclic orientation to each connected component of $U(X)$. That is, we want a mapping $f_\alpha \colon S_X/\sim_\alpha \longrightarrow \text{Acyc}(X)$. It can be shown that such a mapping exists, and moreover, that it is a bijection.

Thus, if we can identify an efficient method to determine all acyclic orientations, then since each such orientation maps to a unique connected component of $U(X)$, all permutations that generate the same acyclic orientation are functionally equivalent. We will show in Section 3.2 that there is indeed an efficient way to identify all acyclic orientations.

Chapter 3

VFA Algorithms

3.1 Update Graph

3.1.1 Generation of Functionally Equivalent Permutations

Given a GDS map F_π for an update sequence π, this algorithm (described in Algorithm 1) computes all update sequences that are functionally equivalent to π using only the dependency graph X (not the vertex functions nor the update scheme). We explain the algorithm with the help of an example. Let a 4-vertex circle graph (as shown in Figure 2.3) represent the dependency graph X for a GDS $\mathcal{S}(X, F, \mathcal{W}, K)$. Let $\pi = (1, 3, 2, 4)$ denote an update sequence. Our goal is to generate the set of permutations that are functionally equivalent to π. To achieve this, we use two data structures: a set and a queue denoted by $derPermsSet$ and $q\_DerPerms$ respectively. Both $derPermsSet$ and $q\_DerPerms$ store all unique permutations that are functionally equivalent to π. Permutations in $derPermsSet$ are kept sorted by their PIDs, when Algorithm 1 is implemented on Hadoop MapReduce framework. The algorithm starts with processing pairs of vertices that are adjacent to each other in π i.e., the pairs (1,3), (3,2) and (2,4). For each of these pairs, it is checked whether there is an edge between their vertices in X. If there is no edge between a pair of vertices, those two vertices are swapped in π to produce a new permutation σ that is functionally equivalent to π. Algorithm 2 is used to perform this swapping. Thus, the pairs (1,3) and (2,4) are swapped in π to generate the permutations $\pi_1 =(3, 1, 2, 4)$ and $\pi_2 = (1, 3, 4, 2)$. We insert π_1 and π_2 into $q\_DerPerms$ and $derPermsSet$. We then extract each permutation in $q\_DerPerms$, one at a time, and process it as was done for π before.

Algorithm 1: Generate all functionally equivalent permutations from a given permutation, using vertex flips. Section 3.1.1 describes this algorithm in detail.

input : A permutation index $parent\_PID$ that uniquely identifies a permutation of vertices and size n of the GDS dependency graph and dependency graph X.

output: A set of permutation indices which are functionally equivalent to the input permutation index $parent\_PID$. These permutations are sorted by their permutation indices within the MapReduce framework.

```
1  //Set derPermsSet contains all permutations that are functionally equivalent to the
   parent_PID.
2  // Queue q_DerPerms stores functionally equivalent permutations in the order in
   which they are derived from the input permutation parent_PID.
3  //Array parentPerm stores the actual permutation of vertices identified by a unique
   permutation index.
4  Instantiate: empty queue q_DerPerms, empty set derPermsSet, permutation index
   swappedPermIndex, Array parentPerm[1..n] ;
5  derPermsSet.add(parent_PID) ;
6  q_DerPerms.enqueue(parent_PID) ;
7  while (!q_DerPerms.empty()) do
8      candidatePermIndex = q_DerPerms.dequeue() ;
9      //getPerm() generates the actual permutation of vertices given a permutation
       index candidatePermIndex.
10     //Algorithm 14 is used to implement getPerm().
11     parentPerm[1..n] = getPerm(candidatePermIndex, n) ;
12     for (j = 1;j ≤ (n − 1);j + +) do
13         if (parentPerm[i],parentPermi[i + 1]) ∉ e[X] then
14             //See Algorithm 2.
15             swappedPermIndex = genSwappedPerm(parentPerm, i) ;
16             if (!derPermsSet.contains(swappedPermIndex)) then
17                 derPermsSet.add(swappedPermIndex) ;
18                 q_DerPerms.enqueue(swappedPermIndex) ;
19 return derPermsSet ;
```

Algorithm 2: Algorithm to generate a new permutation by swapping two adjacent vertices in a given permutation. Algorithm 1 uses this algorithm. Section 3.1.1 describes the use of this algorithm.

input : Array $permArray[1...n]$ storing a permutation of vertices and position i of a vertex in $permArray$.
output: Permutation index.

1 **Instantiate:** Array $swappedPermArray[1....n]$;
2 Swap vertices at position i and $i+1$ of $permArray[]$ to generate the array $swappedPermArray[]$;
3 //$convertToPermIndex()$ returns the unique permutation index that identifies the permutation $swappedPermArray[]$.
4 //Algorithm 15 is used to implement $convertToPermIndex()$.
5 $swappedPermIndex = convertToPermIndex$(swappedPermArray) ;
6 **return** $swappedPermIndex$;

3.1.2 Time and Space Complexity

Let n be the number of nodes in the dependency graph of a GDS and $n_{vfafePIDs}$ be the average number of permutations in a single VFA FEEC, that is computed using update graph-based VFA technique.

Proposition 1 *The time complexity for Algorithm 1 to generate all functionally equivalent permutations from a given permutation, using update graph-based VFA technique, is* $O(n^3 \cdot n!)$.

Proof. We assume that insertion of elements and finding them in an unordered set data structure take constant time. The set $derPermsSet$ contains $n_{vfafePIDs}$ elements in it. The functions $add()$ and $contains()$ take constant time to add and find PIDs, respectively, in this set when it is unordered. When $derPermsSet$ is implemented in Hadoop as an ordered set, the functions $add()$ and $contains()$ take $O(log(n_{vfafePIDs}))$ time. The queue $q\_DerPerms$ contains $n_{vfafePIDs}$ elements in it. We assume that the functions $enqueue()$, $empty()$ and $dequeue()$ take constant time to insert a PID into $q\_DerPerms$, check whether $q\_DerPerms$ is empty and delete a PID from $q\_DerPerms$, respectively. The subroutine $getPerm()$ implemented using Algorithm 14 takes $O(n^2)$ time to generate the actual permutation of vertices $parentPerm$ from a PID $candidatePermIndex$. Subroutine $genSwappedPerm()$ is implemented using Algorithm 2 and takes $O(n^2)$ time to generate $swappedPermIndex$. $genSwappedPerm()$ is called $O(n)$ times for each $parentPerm[]$. The queue $q\_DerPerms$ contains $n_{vfafePIDs}$ number of PIDs and we iterate over these PIDs to generate all permutations that are functionally equivalent to them. Since $n_{vfafePIDs}$ is $O(n!)$ in the worst case, the time complexity for Algorithm 1 is given by $O(n^3 \cdot n!)$. ■

Proposition 2 *The space complexity for Algorithm 1 to generate all functionally equivalent permutations from a given permutation, using update graph-based VFA technique, is* $O(n!)$.

Proof. Data structures $derPermsSet$, $q\_DerPerms$ and $parentPerm[]$ are used in Algorithm 1. $derPermsSet$ and $q\_DerPerms$ require $O(n_{vfafePIDs})$ space to store all permutations of vertices that are functionally equivalent to the permutation $parent\_PID$. The array $parentPerm[]$ requires $O(n)$ space to store an actual permutation of vertices. Subroutines $getPerm()$ and $genSwappedPerm()$ have space complexities of $O(n)$. Since $n_{vfafePIDs} \geq n$ and $n_{vfafePIDs} = O(n!)$, the space complexity for Algorithm 1 is $O(n!)$. ■

3.2 Acyclic Orientations

3.2.1 Generation of Canonical Permutation

For a given GDS, permutations of vertices having the same acyclic orientation are functionally equivalent. We use Algorithm 3 to generate the acyclic orientation and the canonical permutation for a given permutation of vertices.

We explain the algorithm through the following example. We consider a 4 vertex circle graph (as showm in Figure 2.3) as the dependency graph X of a GDS. Let us assume that we want to find the acyclic orientation for two distinct permutations of vertices, given by $\pi = (1, 3, 4, 2)$ and $\sigma = (1, 3, 2, 4)$. Following Algorithm 4, the acyclic orientation for both π and σ will be the directed graph seen in Figure 3.1. Since both of these permutations have the same acyclic orientation, we conclude that they belong to the same functional equivalence class. We then generate a canonical permutation that represents this acyclic orientation.

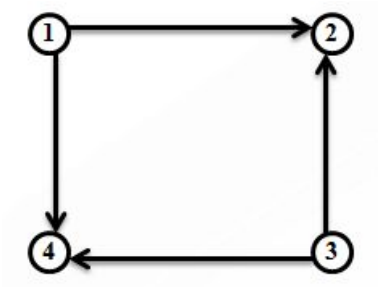


Figure 3.1: Directed graph representing the acyclic orientation for two update sequences $\pi =$ (1,3,2,4) and $\sigma =$ (1,3,4,2) for a GDS with dependency graph $Circle_4$. This directed graph is obtained using Algorithm 4.

The canonical permutation for π and σ is obtained as follows. We iterate over vertices of the directed graph (Figure 3.1) until the graph is empty. In each iteration, we identify those vertices that have zero indegree. We then remove these vertices and all edges incident on

Algorithm 3: Given a permutation index PID generate its canonical permutation. Section 3.2.1 describes this algorithm in detail.

input : Permutation index PID and number of nodes n in the GDS dependency graph X.
output: Permutation index *cannonPermIndx* for the canonical permutation of PID.

```
1  //indegree[] stores the indegree for each vertex of the directed graph diGraph.
2  // zeroInDegNodes stores those vertices of diGraph that have zero indegree.
3  Instantiate: empty graph diGraph to represent the acyclic orientation for PID (diGraph
   represented as an adjacency list), empty array indegree[1..n], Array cannonPerm[1...n] and
   empty queue zeroInDegNodes ;
4  //Generate acyclic orientation of input PID. Refer to Algorithm 4.
5  (diGraph, indegree[]) = generateAcyclOrientation(PID, n, X) ;
6  count = 0, j = 0 ;
7  //Fetch node(s) of diGraph that has (have) zero indegree and enqueue them in
   zeroInDegNodes
8  for (i = 1; i ≤ n; i++) do
9      if (indegree[i] == 0) then
10         zeroInDegNodes.push(i) ;
11         //Update indegree of ith vertex so that it is not processed further.
12         indegree[i]−− ;
13         count++ ;
14         cannonPerm[j++] = i ;
15 Sort elements located between index 1 and count for array cannonPerm ;
16 prevCount = count ;
17 while (!zeroInDegNodes.empty()) do
18     //Remove a node with zero indegree from the diGraph and decrease indegrees of all of
       its adjacent vertices in diGraph.
19     node_0 = zeroInDegNodes.dequeue() ;
20     forall the (vertices v adjacent to vertex node_0 in diGraph) do
21         indegree[v]−− ;
22     Delete node_0 from diGraph ;
23     //Get nodes with zero indegree in updated diGraph and enqueue them in
       zeroInDegNodes.
24     for (i = 1; i ≤ n; i++) do
25         if (indegree[i] == 0) then
26             zeroInDegNodes.push(i) ;
27             indegree[i]−− ;
28             count++ ;
29             cannonPerm[j++] = i ;
30     Sort elements located between index prevCount and count for array cannonPerm ;
31     prevCount = count ;
32 //Algorithm 15.
33 cannonPermIndx = convertToPermIndex(cannonPerm, n) ;
34 return cannonPermIndx ;
```

Algorithm 4: Algorithm to generate the acyclic orientation for a permutation of vertices. Algorithm 3 uses this algorithm. Section 3.2.1 discusses this algorithm in detail.

input : Permutation index PID, number of vertices n in the dependency graph X.
output: Directed graph *diGraph*, Array $indegree[1...n]$.

```
1  Instantiate: Empty graph diGraph representing the acyclic orientation for input
   PID (diGraph represented as an adjacency list), Array perm[1....n], empty Array
   indegree[1..n] ;
2  Set all elements of indegree[] to zero ;
3  //Algorithm 14.
4  perm = getPerm(PID, n) ;
5  for (i = 1; i < n; i++) do
6      for (j = i + 1; j ≤ n; j++) do
7          if (perm[i], perm[j]) ∈ e[X] then
8              //Add directed edge with tail: permArr[i] and head : permArr[j].
9              addDirEdge(diGraph, perm[i], perm[j]) ;
10             indegree[perm[j]]++ ;
11 return diGraph and indegree[] ;
```

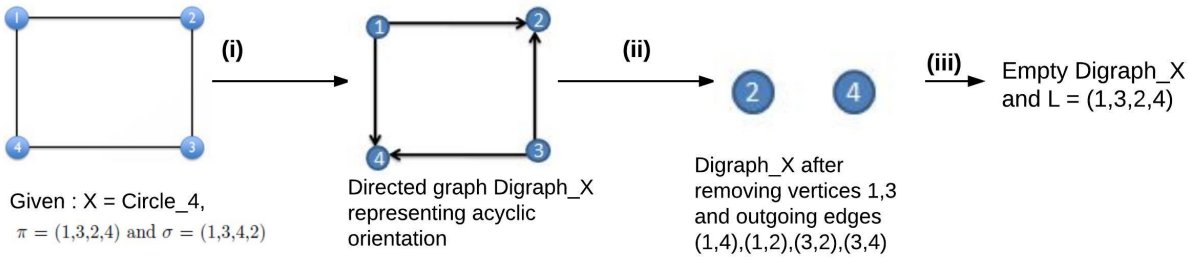
them from the directed graph. These vertices are ordered by their indices and stored in a set L_i (where i represents the ith iteration). The same procedure is repeated in all subsequent iterations until the digraph is empty. In our example, vertices 1 and 3 and edges (1,2), (1,4), (3,2), (3,4) are removed from the digraph in Figure 3.1, in the first iteration. Thus L_1 = (1,3). In the second iteration, we remove vertices 2 and 4 and store them in L_2 as L_2 = (2,4). Combining L_1 and L_2, we get the canonical permutation for π and σ as $(1, 3, 2, 4)$. Figure 3.2 illustrates how the canonical permutation corresponding to π and σ can be generated from their acyclic orientation.

3.2.2 Time and Space Complexity

Let n be the number of vertices and n_e be the number of edges in the dependency graph X of a GDS.

Proposition 3 *Time complexity for Algorithm 3 to generate a canonical permutation from a given PID is* $O(n^2)$.

Proof. We assume that the directed graph representing the acyclic orientation for a given permutation of vertices is represented as an adjacency list, and that adding and removing edges from this directed graph take constant time. Subroutine *generateAcyclOrientation*()



Step (i) Generate directed graph representing acyclic orientation of given permutations (1,3,2,4) and (1,3,4,2)
Step (ii) Remove vertices 1,3 with zero indegree . Remove edges incident on these vertices. Store (1,3) in set L_1, in sorted order
Step (iii) Remove vertices 2,4 from digraph and store them in L_2, in sorted order
Step (iv) Digraph empty. So combine L_1 and L_2 to obtain L as L=(1,3,2,4).

Figure 3.2: Example showing how an acyclic orientation and its corresponding canonical permutation is obtained using Algorithm 3.

generates this directed graph and is implemented using Algorithm 4. $generateAcyclOrientation()$ calls $getPerm()$ to generate the actual permutation $perm[]$ of vertices from the given PID. The time complexity for $getPerm()$ is $O(n^2)$. Thereafter, $generateAcyclOrientation()$ performs $O(n^2)$ computations to generate $diGraph$. Thus the time complexity for $generateAcyclOrientation()$ is $O(n^2)$. $diGraph$ has n vertices and n_e edges. After $diGraph$ is obtained, the canonical permutation for PID is generated by iterating over all vertices in $diGraph$. In each iteration, all vertices with zero indegree are removed from $diGraph$, together with the edges incident on them. Vertices with zero indegree are obtained by scannning through array $indegree[]$, which has $O(n)$ elements in it. Hence, generation of $cannonPerm[]$ from $diGraph$ has a time complexity of $O(n^2 + n_e)$. Subroutine $convertToPermIndex()$ computes that PID which denotes the permutation of vertices $cannonPerm$. $convertToPermIndex()$ has a time complexity of $O(n^2)$. Since the number of edges n_e in the dependency graph X is $O(n^2)$ in the worst case, the total time complexity for Algorithm 3 is given by $O(n^2)$. ■

Proposition 4 *Space complexity for Algorithm 3 to generate a canonical permutation from a given PID is* $O(n^2)$.

Proof. Data structures $indegree[]$, $cannonPerm[]$ and $zeroInDegNodes$ store $O(n)$ elements. $diGraph$ is stored as an adjacency list and has a space complexity of $O(n+n_e)$. Subroutine $generateAcyclOrientation()$ requires $O(n+n_e)$ space to store $diGraph$ and $perm[]$. Since the number of edges n_e in the dependency graph X is $O(n^2)$ in the worst case, the total space complexity for Algorithm 3 is given by $O(n^2)$. ■

Chapter 4

Formal Problem Statement

Given: A GDS; that is, a (directed) dependency graph X, a vertex state set K, a sequence of vertex functions $F = (f)_{v=1}^{n}$, and an update scheme W .

Required:

1. For each update sequence, the phase space of the GDS map;
2. Across all update sequences, the functionally equivalent equivalence classes (FEECs); and
3. Across all update sequences, the cycle equivalent equivalence classes (CEECs).

Chapter 5

Related Work

5.1 Tools for Discrete Dynamical Systems

In this section, we discuss various discrete dynamical systems software that are available, other than our GDS Evaluator tool. These tools have been implemented using serial code. To identify these tools and their features, we first introduce some terminologies.

A **forward trajectory** is a sequence of state transitions, starting from a specified state, and continuing until a limit cycle is reached, or, in the case of probabilistic models, until some number of state transitions are completed.

There are multiple definitions for **asynchronous** update. We have defined asynchronous update earlier in Section 2.1. In [28, 63], asynchronous update is defined as the process of selecting one vertex (at random) whose state is updated at each time t. A single vertex is typically selected uniformly at random, but other approaches are used, such as specifying $b \leq n$ vertices randomly at each time step to update. In [71, 27], a different definition is used: asynchronous update is characterized by each vertex having a different time interval between state updates.

Cellular automata (CA) are dynamical systems that use grid structures of cells. Each cell has a state and is influenced by its nearest neighbors (either 4—north, south, east, and west—or 8—where diagonal cells are included) [79]. A vertex in a dependency graph, which can have connections to any other vertices in the graph, is a generalization of the connectivity of a cell in a grid. As originally conceived, the cell state space was Boolean and the update scheme was synchronous. A cell's next state is computed using the local rule of a cell, which is analogous to a vertex function.

A **Boolean network** (BN), as originally conceived, is a dynamical system similar to a CA except that the regular grid is replaced by a graph. Each vertex serves the role of a cell of a CA, and is connected to any number of the n graph vertices (including self-loops). Each

vertex's function is based on the number of edges incident on it. As the name implies, the vertex state of these networks is Boolean.

A **random Boolean network** (RBN) is random in two senses. Firstly, the dependency graph (also called a **wiring diagram**) consists of directed edges (u, v) implying that the state of vertex u is an input to the vertex function for v. All edges in the graph are specified randomly, but each vertex has the same in-degree d^{in}, and hence the same number of function arguments. Secondly, the vertex function of each vertex is assigned randomly. Once these functions are assigned, they remain fixed throughout computations of dynamics [40]. A common approach is to use elementary cellular automata (ECA) rules, where each function has three inputs [53]. There are many variants of this basic model [28]. Once the graph and vertex functions are assigned, the dynamics are subsequently deterministic.

A **probabilistic Boolean network** (PBN) is a dynamical system which has multiple vertex functions specified for each vertex. At each time (in a forward trajectory), one function is selected for execution according to its associated probability [66]. The selected functions may be executed sequentially or synchronously.

In a **polynomial dynamical system** (PDS), each vertex function consists of polynomials in the vertex states (operations are typically addition, multiplication, and exponentiation) [68].

These types of systems, though some of the most common, are not the only types of dynamical systems. We can define other dynamical systems to suit particular needs.

Table 5.1 summarizes selected properties of some of the tools discussed herein. We first describe some stand-alone (desktop) tools that model CA, BNs, and RBNs.

Mathematica [2] is a tool with CA capabilities. Tools such as Dynamica [48] have been built on top of Mathematica. The Ptolemy project at UC-Berkeley has developed software to model large physical systems [60] used in signal processing, telecommunications, network design, investment management. It has capabilities to analyze both continuous and discrete systems [50]. FiatLux [24] is a CA simulator which sits on top of Ptolemy. We can use this tool to study system robustness. It provides ten different graph types and 12 dynamics models which are applied uniformly to vertices. It supports both sequential and synchronous update schemes.

DDLab [80, 81] is a CA and RBN tool which can evaluate a variety of functions, including elementary cellular automata (ECA) rules. It allows for both sequential and synchronous update schemes. A discriminating and impactful feature of this stand-alone code is the plots that it generates. To the best of our knowledge, its visualizations are well beyond those of most other systems and include state-time and Derrida plots, as well as 3-dimensional graphics. It has additional features, such as the ability to determine predecessors of particular states; to run some dynamical systems backwards; to compute functional and cycle equivalence on CA; and to compute attractor graphs. It can directly compute phase space for networks with 30 or fewer vertices, and can (statistically) compute limit cycles and transients for larger networks.

Table 5.1: Overview of selected dynamical systems software tools that are most closely aligned with our GDS Evaluator tool.

Name	**Vertex State Set Size**	**Complete Phase Space**	**Limit Cycles**	**Update Schemes**	**Number of Vertex Functions**	**Functional Equivalence**	**Cycle Equivalence**	**Type of System**
GSDC	≥ 2	yes	yes	synchronous, sequential, block sequential	15 families, all deterministic	yes	yes	online collaborative environment
ADAM [35]	≥ 2	yes	yes	synchronous, sequential (for PDS only)	user-specified polynomials with logical operations; deterministic and stochastic	no	no	online individual user
FiatLux [24]	2	no	no	synchronous, sequential	12	no	no	individual user; Java app.
DDLab [81]	≥ 2	yes	yes	synchronous, sequential	many (tabular data)	yes, in CA	yes, in CA	individual user; C
BNS [21]	2	no	yes	synchronous	many	NA	NA	individual user
RBNLab [30]	≥ 2	yes	yes	various sequential schemes	many	no	no	individual user; Java app
Matlab RBN [1]	2	yes	yes	synchronous, asynchronous, sequential	many (tabular data)	no	no	individual user; Matlab
BoolNet R Package [59]	2	yes	yes	synchronous, sequential	many (tabular data)	no	no	run within R

BNS (Boolean Networks with Synchronous update) [21] uses update functions that are given as truth tables. The size of each truth table is however exponential in the number of inputs to the function. BNS can compute attractors (i.e., limit cycles), using a technique that does not compute all state transitions.

[1] is an RBN toolkit for Matlab that can handle both synchronous and sequential update schemes. It has significant visualization capabilities and uses tables to specify vertex functions. Furthermore, it implements many variants of the RBN model described in [28]. Another RBN software is RBNLab [30]. It, too, implements a wide range of RBNs, well beyond the classic RBN described earlier [29]. Our GDS Evaluator tool has capabilities to implement CA and BNs. It can also execute some RBNs, but the specification for the random edges and the assignment of the random functions must be done outside the GDS Evaluator tool.

There are other stand-alone software systems that focus on PBNs. There are toolkits within Matlab [3] that are useful in analyzing particular dynamical systems; e.g., the Probabilistic Boolean Network toolkit developed by I. Shmulevich's research team [65]. The BoolNet tool [59] is a package that works within the R statistical software. It evaluates synchronous and asynchronous Boolean networks and PBNs. Vertex functions are based on logical rules (using AND, OR, and NOT). Interestingly, it will also (re)construct (approximately) a graph that gives rise to a specified time series of state transitions. GDSC currently does not implement PBNs; we have a different software tool for stochastic systems.

All tools mentioned thus far are open-source software packages, or add-ons to commercial products, that run on a user's machine, and in this sense are stand-alone applications.

ADAM [35] is a web-based application. It is tailored for biological networks, but can work with other types of networks. Among its models are Petri nets and PBNs. Its novelty, with respect to dynamical systems, is the use of PDSs, where vertex functions are polynomials in the vertex states. PDSs use both sequential and synchronous update schemes. Functions are entered in symbolic format, so there is considerable flexibility in vertex functions (as long as they are polynomial in the inputs). For PDS, ADAM computes the entire phase space for graphs up to 20 vertices, and computes only limit cycles for larger graphs. However, the system supports about 100 state values for each node. Their BN permits synchronous update; the vertex functions are combinations of logical operators (AND, OR, NOT). The entire phase space is computed for BNs.

GINsim [31, 15, 18] is a Java application for simulating (regulatory) networks that use two types of graphs: logical regulatory graphs and state transition graphs for the analysis of logical models. The vertex state set can be of any size, based on the number of expression levels [31]. GINsim can perform asynchronous and synchronous updates with multivalued logical functions.

Cell Collective [33] is a web-based tool that simulates biochemical processes. It computes forward trajectories of successive system states from a provided initial state. It promotes

collaboration among scientists for building large scale biological models (e.g., models with thousands of nodes). There are several features, such as a Knowledge Base that includes a model repository for public use, and Bio-Logic Builder to build models, that enable researchers to explore different models and test hypotheses. The Cell Collective uses the ChemChains simulation engine [34].

We briefly mention several other tools. CellNetAnalyzer (CNA) [42] is a MATLAB toolbox for understanding structural and functional properties of metabolic, signaling, and regulatory networks. CNA is the extension of FluxAnalyzer [43] (originally developed for metabolic network and pathway analysis). MaBoSS [69] is a C++ tool that models biological networks using continuous time Markov processes applied on a Boolean state space. MaBoSS provides a high level language to describe Boolean equations. JCASim [26] is a general-purpose Java simulator in Java. It supports different lattice structures (1-D, 2-D square, hexagonal, triangular, 3-D), neighborhoods, and boundary conditions, and can display the cells using colors, text, or icons. NetBuilder [77] offers capabilities to simulate and analyze regulatory networks. It represents networks using Petri nets and uses a genetic algorithm to evolve genetic regulatory networks (GRNs) needed for specific behavior. Pybool [62] is a Python-based tool for simulation. It helps biologists to define restrictions and conditions to limit the space of networks to ones that are most promising for further experimentation. Pybool uses the IPython package for parallelization. Golly [72] is a tool that simulates Conway's Game of Life and many other types of cellular automata. Ready [36] is a tool that is used for continuous and discrete cellular automata. Ready supports 1D, 2D, 3D, polygonal and polyhedral meshes. Ready relies on OpenCL [70] as a computation engine.

5.2 Hadoop MapReduce

The Hadoop MapReduce framework can be used to solve numerous problems. Reference [16] demonstrates that this framework is suitable for solving a specific set of graph problems such as enumerating triangles and rectangles, finding trusses, finding components and barycentric clustering, which can be characterized using "independent local communication". The MapReduce framework is also effective for performing data intensive analyses [67, 23] such as High Energy Physics data analyses, and for solving embarassingly parallel scientific computing problems [13, 67] such as generation of independent Gaussian random variables using the Marsaglia polar method. Other data intensive problems in which MapReduce has been effectively employed include performing integer sorting [13] using bucket sort and for designing image conversion tool for social media transmoding and transcoding [41].

Graph algorithms that require depth first traversals do not have efficient analogous MapReduce algorithms [16]. More generally, MapReduce is not well suited for iterative algorithms [13, 23] such as conjugate gradient, Fast Fourier Transform, block tridiagonal and Kmeans clustering. Alternate frameworks such as Pregel [57], HaLoop [12] and Twister [22] are more suitable for these iterative algorithms.

There are many simulation frameworks that use the MapReduce programming model. Reference [64] implements a multi-agent simulation framework on a Hadoop cloud. Each agent in this framework has a unique ID, a timestamp and an agent type associated with it. These properties define the state of an agent. An $update - agent - state$ function is defined for each agent type. This function governs how the state of an agent changes with time. This multi-agent simulation framework is modeled as a series of MapReduce jobs. Each job in this series represents an iteration in the simulation. Each agent is modeled as a separate input file to these jobs. A particular job invokes multiple map tasks, one for each agent. These map tasks execute in parallel and are responsible for updating the states of agents. Reduce tasks write the data back to the file that is associated with each agent.

HiMach [73] is a parallel framework that provides users with a MapReduce style programming interface to write sequential molecular dynamics (MD) trajectory analysis programs. Users can define MD trajectories, conduct per-frame and cross-frame analyses, aggregate intermediate results and launch an analysis job using the HiMach API. The HiMach runtime is responsible for automatic parallel execution of these sequential user programs. It assigns tasks to processors, communicates and exchanges data among processors, issues parallel I/O requests and stores and manages intermediate key-value pairs.

Reference [75] is another work which extends the MapReduce programming model to design a simulation engine, BRACE, that processes behavioral (agent-based) simulation models. Reference [17] uses MapReduce implementation of the Monte Carlo bootstrap algorithm to run simulations for Metabolic Flux Analysis (MFA), which is used in Systems Biology to estimate intracellular reaction rates.

Various works have conducted performance evaluation of the Hadoop MapReduce framework. Reference [39] identifies four factors that affect the performance of MapReduce jobs running on Hadoop. These factors are 1) the I/O mode that map tasks use to read data from different storage systems, 2) the indexing strategy used to speedup data processing, 3) the decoding scheme used to transform raw data to structured records and 4) the key comparison scheme used to sort intermediate key-value pairs. The performance of the Hadoop MapReduce framework can be improved by using direct I/O mode, range-indexing scheme, mutable decoders and fingerprinting-based map-side sorting. Reference [23] observes that the I/O bandwidth of a cluster node limits the performance of a data intensive MapReduce job. The performance of such jobs do not depend on the number of processor cores that are present in a cluster node. This work further shows that the MapReduce framework overhead decreases with an increase in the number of computations of a MapReduce job and in the amount of data processed by a job.

Reference [74] evaluates the Hadoop MapReduce framework using a MapReduce implementation of the Apriori algorithm [7], which is an iterative algorithm to mine frequent datasets in a database. One major finding of this work is that using multiple reducers in the reduce phase introduces framework overheads which is greater than the performance gain achieved by parallelizing the reduce phase. The performance of MapReduce jobs is closely tied to the

scheduling schemes that are used by Hadoop.

Various works [39, 51, 61] focus on studying these schemes, improving them and designing new ones. In heterogeneous environments, such as Amazon's Elastic Compute Cloud, the performance of MapReduce jobs depend on how Hadoop's task scheduler determines when to speculatively re-execute straggler tasks that affect the response time the most. Hadoop's default scheduling algorithm can cause performance degradation of jobs running in heterogeneous clusters. LATE [82] is a robust scheduling algorithm that uses estimated finish times to determine tasks for speculative execution in heterogeneous clusters.

Reference [37] designs a performance model to evaluate how performance of Hadoop in virtualized environments is affected by allocation of virtual machines over physical servers, configuration of a virtual machine and number of concurrent jobs. Disk I/O shared by multiple virtual machines in a physical server introduces virtualization overhead. This overhead impacts the performance of I/O intensive jobs more than that of CPU intensive jobs. I/O intensive jobs can achieve significant performance improvements with an increase in the number of virtual machines and not with an increase in the number of VCPUs in a virtual machine. Decrease in the number of concurrent jobs and wide allocation of virtual machines over physical servers also result in performance gains for I/O intensive jobs.

Chapter 6

New GDS Evaluator Tool

This chapter describes the different features that we have added to the GDS evaluator tool so that it can process more complex dynamical systems. We also give an overview of the different paradigms that we have used to perform GDS evaluation.

6.1 New Features

Following are the new features that we have added to the GDS Evaluator tool.

- **Rule based vertex function assignment**
  Earlier version of GDS evaluator tool allowed static assignment of vertex functions only, i.e. it necessitated assigning vertex functions to its nodes before the start of GDS evaluation. We have added a new feature that allows dynamic assignment of vertex functions. Using this feature, a GDS can be assigned different functions at every time step during the course of GDS evaluation. We can define "rules" to govern such vertex function assignment. One such example of a rule is : *For a Boolean GDS, assign NOR function to a node if the node is in state 1. Else apply NAND function to the node.*

- **Open/Closed Neighborhood**
  As discussed in Chapter 2, we consider the 1-neighborhood of a vertex while computing its next state. The new GDS evaluator tool allows us to specify whether we want to consider the open-neighborhood or closed neighborhood of a vertex while calculating its next state.

- **Multi-component vertex state**
  This feature allows vertices to have a multi-dimensional state. This is achieved by enabling each vertex state to comprise of any number and type of components. Each

such component will represent a different property of the vertex state. For example, the state of a vertex may contain 3 integer components namely x_i, t_d and t_u, where $x_i \in (0, 1)$, t_u represents the up threshold value for the vertex and t_d the down threshold value for the vertex. For each component of the vertex state, we also define a set of "permissible" values that the component can take. It is also possible to define vertex functions that update all/some components of the vertex state.

- **Support for phase space computation with reduced dimension**
  We have modified GDS evaluator tool so as to enable users to perform phase space analysis on a reduced dimension i.e. users can specify which components of the node state are to be included for performing phase space analysis. For example, let us consider that the vertex state of a GDS consists of three components and is represented as a 3 integer state vector (t_d, t_u, x_i) where t_d is the down threshold, t_u is the up-threshold and x_i is in $\{0, 1\}$. Suppose we want to compute the phase space solely on the component x_i, i.e. we want to construct a phase space on system states where each vertex's state is the component x_i only. The new version of GDS Evaluator tool enables us to achieve this. It allows all dimensions of the node state to participate in computing the state transition of a node, while including certain components only to compute the phase space. Components to be included in phase space analysis are to be specified on a per graph basis.

6.2 GDS Evaluation Paradigms

GDS evaluation consists of computing the phase space for each permutation. Then, over all permutations, the functional equivalence classes and cycle equivalence classes are computed. Figure 6.1 shows the different paradigms that we have used to perform GDS evaluation. We discuss each of these paradigms in detail in the subsequent chapters.

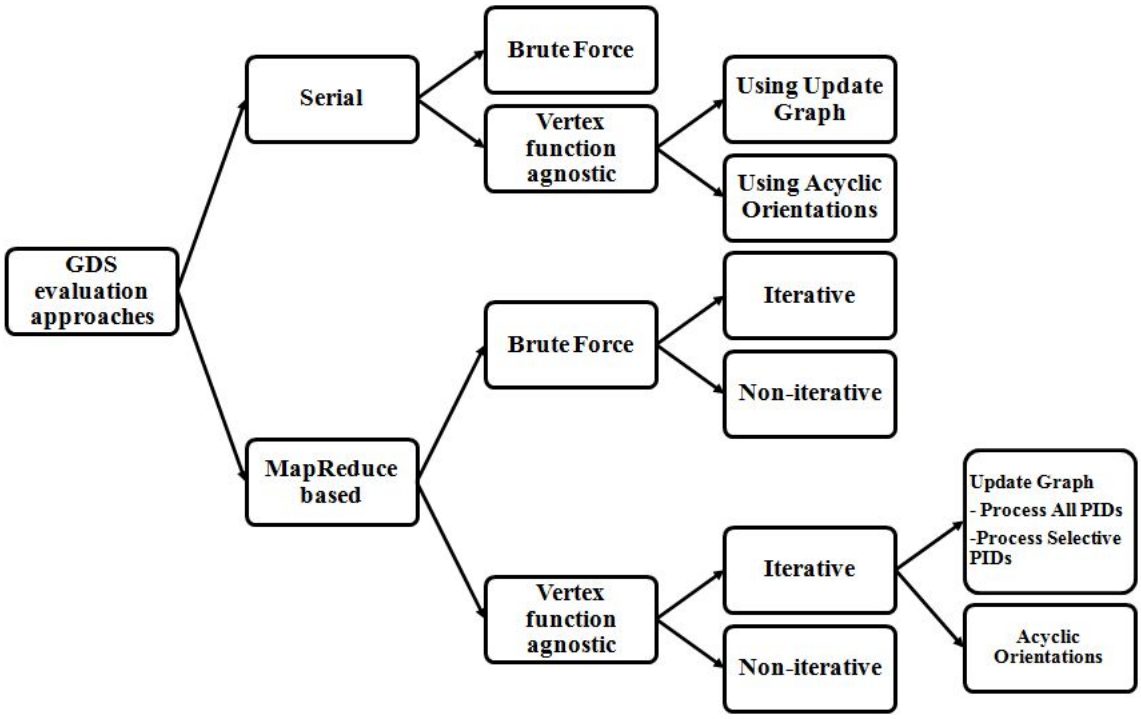


Figure 6.1: Techniques used to perform GDS evaluation.

Chapter 7

Serial GDS Evaluation

This chapter describes different serial algorithms that we have used to perform GDS evaluation. Algorithms 7-13 used in this chapter are in Appendix B.

7.1 Data Structures used in Serial GDS Evaluation Paradigms

In this section, we describe different data structures that we have used across all serial GDS evaluation algorithms.

1. **Permutation map:** This is a map data structure. We denote it as $permMap <PID, perm>$. This map stores all permutations of vertices for a given GDS and indexes these permutations using a unique integer PID. The PID for a permutation and the permutation itself forms a key-value pair in this map. We populate $permMap$ by generating permutations ordered lexicographically and index each such permutation using its lexicographic rank. $permMap$ thus has a space complexity of $O(n \cdot n!)$ for sequential systems and $O(n)$ for synchronous systems.

2. **System state map:** This map data structure stores all possible system states of a GDS and indexes them using a unique integer $SSID$. We denote this map as $sysStMap <SSID, sysState>$, where $sysState$ represents the system state of a GDS. This map is populated using Algorithm 7.

3. **PID-phase space map:** This map stores all possible system state transitions for a single permutation of vertices. A PID and its corresponding phase space ($phaseSpace$) is used as a key-value pair in this map. We denote this map as $pidPhaseSpaceMap <PID, phaseSpace>$ $phaseSpace$ is stored as another map consisting of system state transitions as key-value pairs. Each system state transition can be represented as a key-value pair

$<SSID_1, SSID_2>$, where $SSID_1$ transitions to $SSID_2$. For a Boolean system, PID-phase space map contains $O(2^n)$ records in it.

4. **FEEC-phase Space map:** This map contains the phase space for each functional equivalence class. The map is denoted as $feecPhaseSpaceMap <feec, phaseSpace>$, where $feec$ is an integer that denotes a FEEC and is represented by a PID belonging to that class.

5. **FEEC map:** This data structure maps a FEEC to the set of PIDs belonging to it. We denote this map as $feecMap <feec, PIDSet>$, where $feec$ is a FEEC represented by a PID belonging to that class and $PIDSet$ is the set of PIDs comprising that class.

6. **CEEC map:** This data structure maps each CEEC to the set of PIDs that belong to it. We denote this map as $ceecMap <ceec, PIDSet>$, where $ceec$ is an integer that denotes a CEEC and is represented by a PID belonging to that class and $PIDSet$ is the set of PIDs that form the equivalence class.

7. **PID-cycle structure map:** This maps each CEEC to its cycle structure. Each CEEC is represented using a PID belonging to it. The cycle structure corresponding to a CEEC is the set of cycle lengths, and for each cycle size, the number of such cycles (i.e., the multiplicity) in its phase space. We denote the PID-cycle structure map as $PIDCycleStructMap <PID, cycleStructure>$.

7.2 Brute Force GDS Evaluation

Brute force GDS evaluation algorithm works as follows. We first populate the maps $permMap$ and $sysStMap$, following schemes described in Section 7.1 and Algorithm 7. We then iterate over all PIDs in $permMap$. For each PID (PID) in $permMap$, we loop over all SSIDs in $sysStMap$ to generate all system state transitions corresponding to PID. This gives us the phase space PS for PID. We then determine the FEEC that PID belongs to. This is done by comparing PS with the phase spaces of existing (if any) FEECs. If an exact match is found with an existing equivalence class, then PID is added to that class. If no match is found, then a new equivalence class is created and PID is added to this new class. CEECs are computed by taking one PID from each FEEC and computing the cycle structure for the phase space of this PID. If multiple PIDs belonging to different FEECs have the exact same cycle structure, then all PIDs belonginging to these FEECs are grouped together in the same CEEC. Brute force GDS evaluation on a serial machine is done following Algorithm 8. Algorithm 8 uses Algorithms 9 and 10 to determine the FEEC that a PID belongs to and to determine the CEECs, respectively.

7.3 VFA GDS Evaluation

The VFA GDS evaluation paradigm differs from the brute force GDS evaluation paradigm in the following manner. Unlike in the brute force paradigm, this does not generate the phase space for all possible permutations of vertices. We first apply a suitable VFA technique (as discussed in Section 2.5) to obtain VFA FEECs. Then we take one permutation from each FEEC and generate the phase space for these permutations. If the phase spaces of permutations belonging to different VFA FEECs are exactly same, then these VFA FEECs are further combined to obtain a single FEEC. The procedure to obtain CEECs is the same as that used in brute force GDS evaluation paradigm. Algorithm 11 is used to implement the VFA GDS evaluation paradigm. Algorithm 11 uses Algorithms 12 and 13 to compute FEECs using a VFA technique and to determine whether multiple VFA FEECs can be merged, respectively.

Chapter 8

GDS Evaluation using MapReduce Framework

This chapter describes different MapReduce programming paradigms that we have used to perform GDS evaluation. We discuss each of these paradigms in detail and analyze their space and time complexities. We determine the space complexity of the map phase of a MapReduce job by summing the storage space required by its map tasks and that required by all the key-value pairs produced by all of its map tasks. Similarly, the space complexity for the reduce phase of a MapReduce job is computed by summing the storage space required by its reduce tasks and that required by all the key-value pairs generated by all of its reduce tasks. Our time complexity and space complexity measures depend only on the algorithm, not on the details of how the Hadoop MapReduce framework is implemented. Our primary focus here is sequential dynamical systems. Algorithms 14-35, used in this chapter, are in Appendix B.

8.1 Hadoop MapReduce Framework

Hadoop MapReduce [25] is a software framework that allows a user to write applications which can process vast amounts of data in parallel on large clusters of commodity hardware. This framework uses the classic **MapReduce** programming paradigm to process data. As described in [19], this programming model takes a set of key/value pairs as its input and produces a set of key/value pairs as its output. A user needs to express a computation in terms of two functions: *Map* and *Reduce*. A *Map* function takes an input key/value pair and produces a set of intermediate key/value pairs as its output. These intermediate pairs are sorted and grouped by their key values by the Hadoop framework and then passed on to the *Reduce* function. The *Reduce* function takes an intermediate key I and the set of values correspoding to I, as its input. Thereafter it performs some user-defined computation on

this key I and its associated set of values and yields key/value pairs as the output.

A MapReduce job on Hadoop consists of the input data, the MapReduce program and the configuration information. Typically, the input and output of a MapReduce job is stored on the HDFS, the distributed filesystem for Hadoop. Hadoop runs this job by dividing it into map and reduce tasks. An input file to a MapReduce job is automatically divided into M independent chunks called *input splits*. These chunks are stored on HDFS and are distributed across the nodes of the cluster. The value of M depends on the size of the input file.

Hadoop creates a map task for each split and each map task runs the user-defined map function for each record in its split. Hadoop attempts to run a map task on the same node where an input split resides on HDFS. Output from map tasks are written to local disk and not to HDFS.

The input to each reduce task comes from all map tasks. If we have a single reduce task, then all sorted map outputs are transferred across the network to that node where the reduce task is running. These map outputs are merged and then passed on to the user-defined reduce function. Figure 8.1 shows the data flow with a single reduce task. The number of reduce tasks is independent of the size of the input file, and is specified by the user. When there are multiple reducers, the map tasks partition their key-value pairs such that each reducer task processes a single partition. This partitioning is done using Hadoop's default partitioner, unless a user defines a different partitioning function. The data flow with multiple reducers is illustrated in Figure 8.2.

The Hadoop framework takes care of moving data between map and reduce tasks, scheduling tasks, monitoring them and re-executing them in case of any failure.

Memory Optimizations to model GDS using Hadoop MapReduce

GDS evaluation using serial algorithms is memory-intensive. This is because these algorithms utilize considerable amounts of memory to store all permutations of vertices and all system states of a GDS. The total memory required to store these data structures is $O(n \cdot (n! + |K|^n))$ on a serial machine. While evaluating a GDS using Hadoop, we have used the following memory optimizations in order to overcome the memory overhead associated with serial algorithms.

1. **Permutation map:** We have defined two permutation maps $M_1 \colon \mathbb{N} \longrightarrow \mathbb{N}^n$ and $M_2 \colon \mathbb{N}^n \longrightarrow \mathbb{N}$. The map M_1 generates a permutation of vertices whose lexicographic rank is equal to a given PID. M_2 generates an equivalent PID for a given permutation of vertices. Note that M_1 is a bijection because $M_2 = {M_1}^{-1}$. Algorithms 14 and 15 are used to implement maps M_1 and M_2.
2. **System state map:** We have defined two system state maps $M_3 \colon \mathbb{N} \longrightarrow \mathbb{N}^n$ and

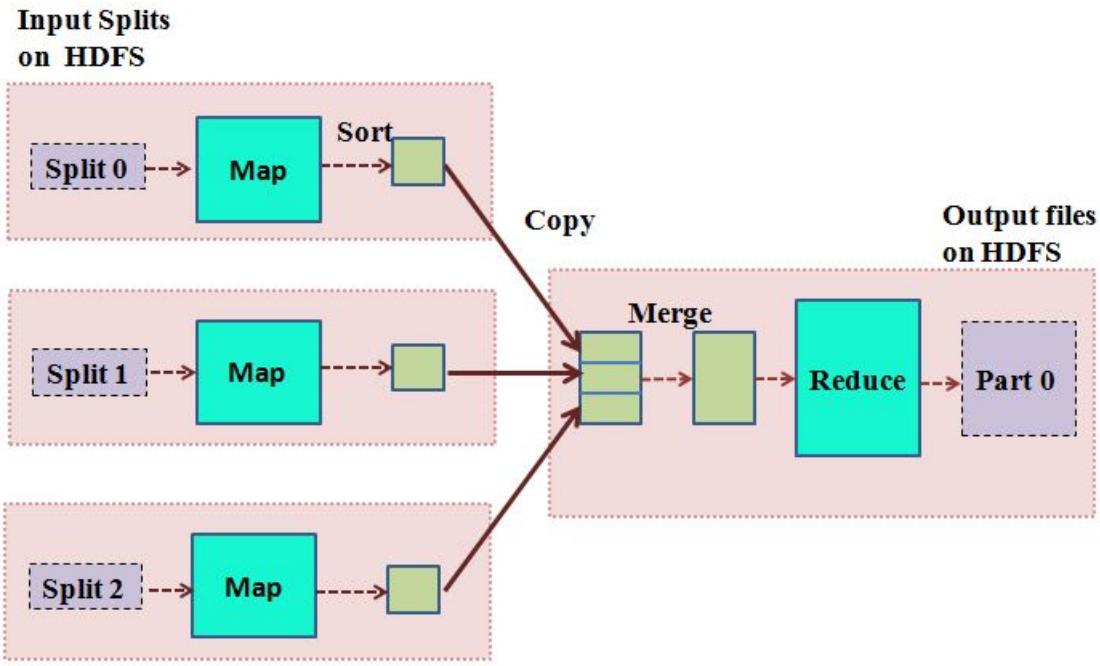


Figure 8.1: MapReduce data flow with single reduce task [78].

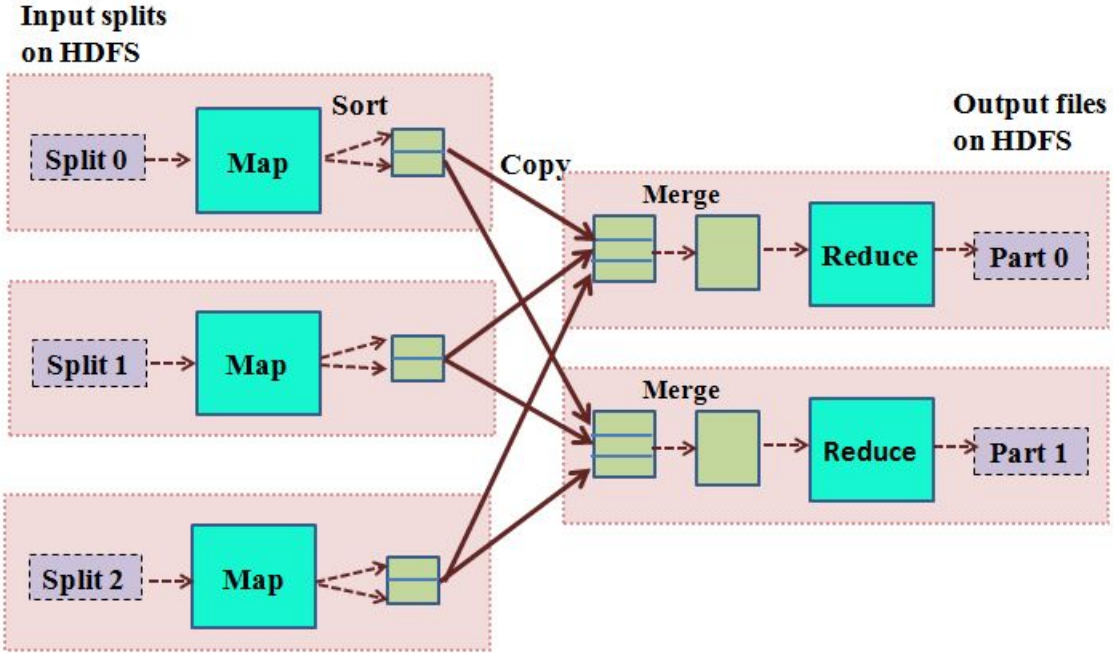


Figure 8.2: MapReduce data flow with multiple reduce tasks [78].

$M_4 : \mathbb{N}^n \longrightarrow \mathbb{N}$. M_3 allows conversion from a SSID to its corresponding system state. M_4 allows conversion from a system state to its equivalent SSID. Hence, M_3 is a bijection with $M_4 = M_3^{-1}$. We use the schemes described in Algorithms 17 and 16 to implement M_3 and M_4 respectively.

These memory optimizations do result in an increase in the number of computations required to evaluate a GDS.

8.2 Brute Force GDS Evaluation

8.2.1 Non-iterative Algorithm

Non-iterative brute force GDS evaluation is performed with Hadoop MapReduce framework by chaining three MapReduce jobs, as shown in Figure 8.3. The output of a job in this chain acts as the input to the next job in the chain. Algorithm 18 shows the overall workflow of non-iterative brute force GDS evaluation. This evaluation paradigm can be used for both sequential and synchronous GDSs.

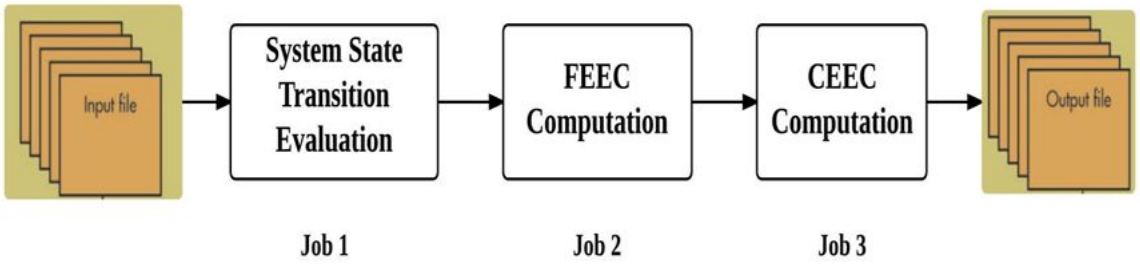


Figure 8.3: Chaining of multiple MapReduce jobs to perform GDS evaluation using non-iterative brute force approach.

The chained MapReduce jobs, used in this evaluation paradigm, are discussed in detail as follows.

8.2.1.1 System State Transition Evaluation Job

This job generates all system state transitions for all permutations of vertices of the GDS. It thus computes $n! \cdot 2^n$ system state transitions for a sequential Boolean dynamical system and 2^n system state transitions for a synchronous Boolean system. All these computations are done in parallel using MapReduce framework. Figure 8.4 provides an overview of the workflow of this job, as described next.

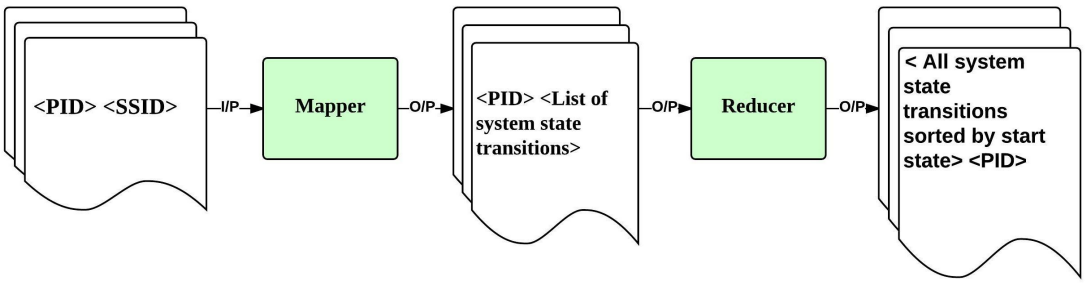


Figure 8.4: Workflow of system state transition evaluation MapReduce job.

The input to this job is a text file having the following format :
$<PID>$ $<SSID>$
where PID is a unique identifier for a permutation of vertices and $SSID$ is a unique identifier for a GDS system state. Hence, an input file for a sequential Boolean system has $n! * 2^n$ lines in it, while that for a synchronous Boolean system has 2^n lines. After the job is submitted to the cluster, the input file is split into M pieces. The value of M depends on the size of the input file. Hadoop spawns M different map tasks, each of which processes one split of the input file. For each line of its input split, a map task passes the line offset and the line as a $<Key, Value>$ pair to the $map()$ function. The $map()$ function is thus called for each line of an input split of a map task.

The $map()$ function extracts the $<PID>$ as the key and $<SSID>$ as the value from the line passed to it. It then computes the next system state index $nextSSID$ that the GDS transitions to and concatenates the $SSID$ and $nextSSID$ values to obtain the $stateTransition$ string. Thereafter it outputs the PID and the $stateTransition$ string as a $<Key, Value>$ pair. The total number of key-value pairs output by all map tasks is $n! \cdot 2^n$ for a sequential Boolean system and 2^n for a synchronous Boolean system. Algorithm 19 describes the $map()$ function of this job. Key-value pairs produced by map tasks are partitioned into R groups. A separate reduce task is spawned for each of these R groups. Each reduce task sorts and groups the $<PID, stateTransition>$ pairs, in its partition, by their PID. Then it iterates over these sorted intermediate data and for each unique PID encountered, it passes the PID and the corresponding list ($SysStateTransList$) of $stateTransition$ values to a $reduce()$ function. The $reduce()$ function extracts each $stateTransition$ from the list $SysStateTransList$. It then parses the initial ($fromSSID$) and final ($toSSID$) system state from each $stateTransition$ string. A Map data structure is used to store these $<fromSSID, toSSID>$ values. This Map is ordered by its key $fromSSID$. The $reduce()$ function iterates over all entries in this Map and concatenates all $toSSID$ values to form the string $sortedStateTrans$. $sortedStateTrans$ thus represents the phase space for a PID and contains all system state transitions sorted by their initial states. The output from each $reduce()$ function is a $<sortedStateTrans, PID>$ key-value pair, which is written to a file on HDFS. The total number of key-value pairs output by all reduce tasks is $n!$ for sequential Boolean systems and 1 for synchronous Boolean systems. Algorithm 20 explains how the

reduce() function works.

8.2.1.2 FEEC Computation Job

This job is used to compute all functional equivalence classes for the given GDS and is Job 2 in Figure 8.3. The workflow for this job is shown in Figure 8.5

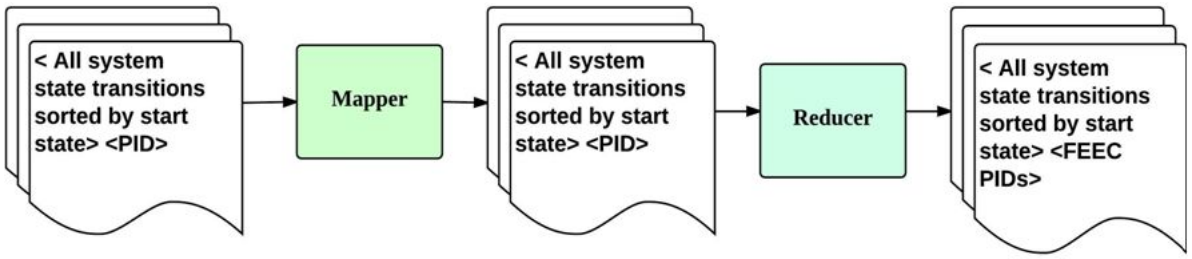


Figure 8.5: Workflow of FEEC computation MapReduce job.

The output from the system state transition evaluation job is passed as an input to this job. Thus, each line of the input file to this job has two entries, the first represents the phase space (*phaseSpace*) of a PID and second is the PID itself. The *phaseSpace* contains $|K|^n$ SSIDs. The map task of this job ouputs these phase spaces and PIDs as key-value pairs. Algorithm 21 describes how the *map*() function works.

Each *reduce*() function is passed a phase space and the corresponding set of PIDs with the same phase space. Hence, all PIDs in this set belong to the same functional equivalence class. The *reduce*() function outputs the phase space of a functional equivalence class together with the set of PIDs in that class. Algorithm 22 explains the *reduce*() function in detail. For a sequential Boolean system, this job takes $n!$ key-value pairs as its input and generates $n!$ and n_{FE} key-value pairs from its map and reduce phases respectively. (n_{FE} is the number of FEECs in a given GDS.)

8.2.1.3 CEEC Computation Job

This job is used to compute cycle equivalence classes of the given GDS. Figure 8.6 shows the workflow of this job.

The output of FEEC computation job is fed as the input to this job. Thus the input file to this job has the following format:
< Phase Space of a FEEC > < Set of PIDs belonging to the same FEEC >
Each *map*() function of this job processes a single line of the above input file. It extracts the phase space structure of a FEEC together with the set of PIDs in that class. It then parses out one PID from this FEEC and computes the cycle structure of this PID from the

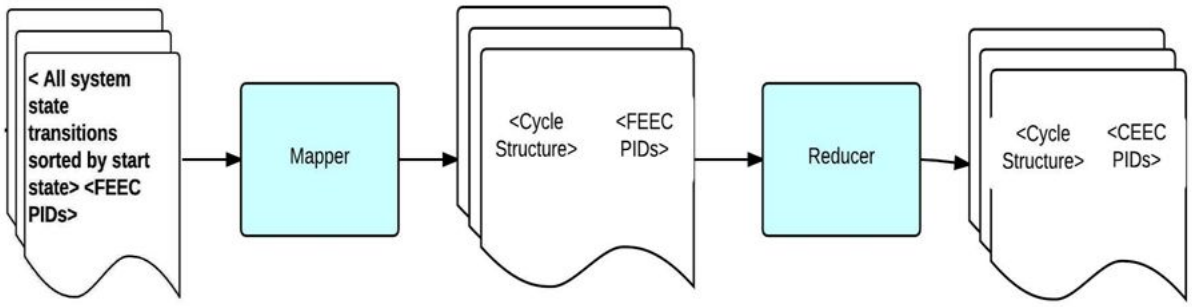


Figure 8.6: Workflow of CEEC computation MapReduce job.

strongly connected components of its phase space graph. All PIDs belonging to the same functional equivalence class have the exact same cycle structure. The *map*() function thus outputs, as key-value pairs, the cycle structure of a FEEC and the set of PIDs belonging to the same FEEC. The cycle strcuture is the set of cycle lengths, and for each cycle size, the number of such cycles (i.e., the multiplicity) in the phase space. Algorithm 23 describes the *map*() function.

Each *reduce*() function receives a unique cycle structure and a set of permutations having the same cycle structure. It then concatenates all these permutations together to form a single cycle equivalence class. The cycle structure of a CEEC and the permutations belonging to it, are emitted by the *reduce*() function as key-value pairs. Algorithm 24 illustrates how the *reduce*() function works. The CEEC computation job takes n_{FE} key-value pairs as its input and produces n_{CE} key-value pairs as its output. (n_{CE} is the number of cycle equivalence classes for a given GDS.)

8.2.2 System State-based Iterative Algorithm

This algorithm, like that of Section 8.2.1, uses a sequence of MapReduce jobs chained to one another. One of the jobs in this chain is, however, run iteratively. As seen in Figure 8.7, GDS evaluation starts with execution of iterative MapReduce jobs that compute functional equivalence classes without a priori computing phase spaces. This is followed by phase space computation job that generates the phase space for each functional equivalence class. Finally, the CEEC computation job computes the cycle equivalence classes. Algorithms 25 and 26 give an outline of how system state-based iterative brute force GDS evaluation works.

The chained MapReduce jobs, used in this evaluation paradigm, are discussed in detail as follows.

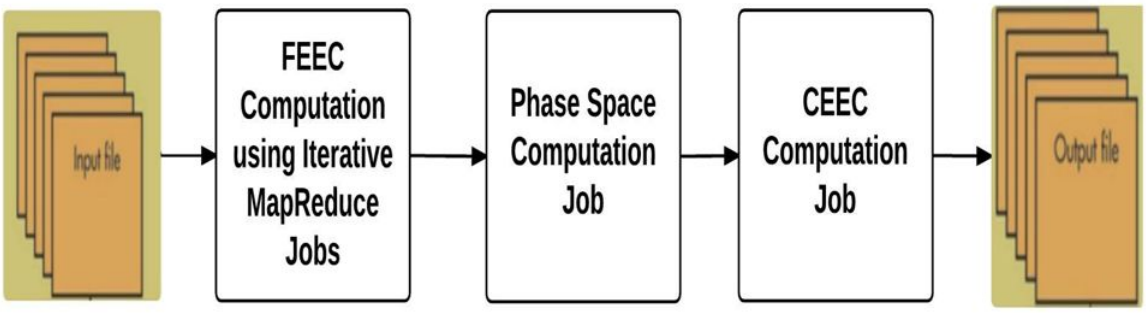


Figure 8.7: Chaining of multiple MapReduce jobs to perform brute force GDS evaluation using system state-based iterative algorithm.

8.2.2.1 Iterative FEEC Computation MapReduce jobs

The general idea for iterative MapReduce jobs is to chain multiple identical jobs together, using the output of the last one as an input to the next one. For a given GDS, iterative FEEC computation jobs run at most $|K|^n$ number of times, which is the total number of system states of a GDS. Here n and K denote the number of vertices and the vertex state set of the GDS respectively. Hence we need to run at most $|2|^n$ instances of these iterative jobs to compute FEECs for a Boolean system. In each iteration, we process a particular GDS system state.

We use an example to illustrate how iterative MapReduce jobs can be used to compute functional equivalence classes. The first iteration works as follows. Each line of the input file for the first iteration has the following entries:
$< PID > < SSID_0 >$
where PID is a permutation index that uniquely identifies a permutation of vertices and $SSID_0$ is 0 and denotes that GDS system state which is encoded as 0. Thus for a sequential GDS, the input file for the first iteration will have $n!$ lines. An example of this input file is shown in Figure 8.8. Each $map()$ function of this job parses a line of this input file and

0	0
1	0
2	0
..	..
$(n! - 1)$	0

Figure 8.8: Example of input file for first iteration of FEEC computation MapReduce jobs. The first column denotes a PID and the second column denotes system state index 0.

extracts a $< PID, SSID_0 >$ pair. Assuming 0 to be an initial state of the GDS, the $map()$ function computes the next state ($SSID_{to}$) that the GDS will transition to. It then emits this state transition and the PID as a key-value pair. The state transition is represented

as a string obtained by concatenating $SSID_0$ and $SSID_{to}$. Figure 8.9 shows an example of the intermediate key-value pairs output by a $map()$ function. The first column in this figure is a state transition string and the second column is a PID. Figure 8.10 is how this intermediate output looks like after the MapReduce framework shuffles and sorts the output (seen in Figure 8.9) by their key values. The first column in this figure is a system state transition and the second is a list of PIDs with the same system state transition.

<SSID$_0$ SSID$_{to}$>	<PID>
0 1	0
0 2	1
0 1	7
0 5	4
0 2	3
..	..

Figure 8.9: Example of intermediate map output key-value pairs from the first iteration of an iterative FEEC computation job, before shuffle and sort by Hadoop. The first column (i.e., key) in this figure is a state transition string and the second column (i.e., value) is a PID. Hence the first line of this file indicates that the GDS has a state transition from state index 0 to state index 1 for PID 0.

<SSID$_0$ SSID$_{to}$>	<PID>
0 1	0 7
0 2	1 3
0 5	4
..	..

Figure 8.10: Example of intermediate map output key-value pairs from first iteration of iterative FEEC computation job, after shuffled and sorted by Hadoop. The first column in this figure is a system state transition and the second is a list of PIDs with the same state transition. Hence the first record indicates that the GDS transitions from state index 0 to state index 1 for both of the PIDs 0 and 7.

Each $reduce()$ function receives a system state transition string and a list $pidList$ of PIDs having the same system state transition. If $pidList$ contains only one PID, then the reduce function dumps this PID to a file on HDFS and does not process it in subsequent iterations. All such dumped PIDs will be referred to as *discarded* PIDs in the remaining document. This is done because a single PID in $pidList$ indicates that there are no other permutation indices which have the exact same state transition from state 0, as this PID. Thus a *discarded* PID forms a functional equivalence class by itself. If a $pidList$ contains multiple PIDs in it, then the $reduce$ function will substitute the contents of the system state transition string by a PID $\in pidList$. This not only decreases the length of the system state transition string but

also enables encoding the state transition computed in the first iteration using a unique PID. At this point in the computations, the permutations that will be processed further are those that may produce the same GDS map as other permutations. The *reduce*() then calculates the value of $SSID$ that is to be processed by the next iteration and appends this value to the system state transition string. The system state transition string can, thus, be thought of as consisting of the following elements:
<A PID representing the group of PIDs that have the exact same system state transition in the first iteration> and <$SSID$ to be processed in the next iteration>.
In our implementation of these iterative jobs, we process system states in ascending order of their indices. Thus system state with index 0 is processed in the first iteration, that with index 1 is processed in the second iteration, and so on. The *reduce*() function outputs the *pidList* and the updated system state transition string as a key-value pair. Each *reduce*() function increments the value of a global counter if it receives a *pidList* that contains mulitple PIDs in it. The value of this counter is checked at the end of every iteration in order to determine whether we proceed to the next iteration. We continue with the next iteration if there is at least one reducer output key which consists of multiple permutations. If all reducer output keys consist of a single PID, then we exit from the loop and continue with the phase space computation job. This is done because a single PID in all of the reducer output keys implies that each of them forms an equivalence class by itself and hence we do not need to process them further. This process will terminate, as shown in Algorithm 26 because it will execute at most $|K|^n$ times. This will only be the case for two or more permutations that are functionally equivalent. Figure 8.11 shows the output from a reduce task, at the end of the first iteration.

<PIDs>	<State transition history $SSID_1$>
0 7	0 1
1 3	1 1

Figure 8.11: Example of output file from first iteration of FEEC computation MapReduce jobs. The first column contains a string of PIDs. The second column has two elements: the state transition history and the SSID to be processed in the next iteration. For example, the first record indicates that PIDs 0 and 7 have the exact same state transition from system state 0. This state transition is encoded using PID 0. Hence we see 0 as the first element of the second column. The second element of the second column is 1 which denotes that system state with index 1 will be processed in the next iteration. We note that there is no record in this file corresponding to map output key-value pair <05,4>; see Figure 8.10. This is because there is no other PID, apart from PID 4, which transitions from state 0 to state 5. Hence PID 4 is discarded in the reduce phase and is not an input for iteration 2. Discarding PIDs leads to computational efficiencies.

The second iteration works as follows. Input to the job run in the second iteration comes from the output of the reduce phase of first iteration. We can consider each line of input to

this job to have the following format:
<PID String> <State Transition History> <SSID to be processed in iteration 2>,
where <PID String> is a set of PIDs having the exact same state transition in iteration 1. <State Transition History> indicates the state transition computed in iteration 1 for a given <PID String>. As explained above, this history is reprsented using a PID $\in$ <PID String>. The SSID that is processed in iteration 2 is 1. Hence all *map*() inputs will have a value of 1 for this field. Figure 8.11 is an example of the input file for the MapReduce job run in the second iteration. Each *map*() function of this job processes a line of this input file and extracts the PID string, state transition history and SSID fields. For each PID in <PID String>, it does the following:

1. Compute the next system state *toSysState* that the GDS will transition to from system state index 1.
2. Append *toSysState* to state transition history.
3. Output the new state transition history and PID as a key-value pair.

Figures 8.12 and 8.13 show output from the map phase and the reduce phase of the job run in the second iteration, assuming that the input to this job is the file shown in Figure 8.11.

<State transition history $SSID_{to}$>	<PIDs>
0 5	0 7
1 7	1
1 3	3
..	..

Figure 8.12: Example of intermediate output from the map phase of the second iteration of the FEEC computation job, after the output has been shuffled and sorted by Hadoop MapReduce framework. The first column (i.e., an intermediate key) has the state transition history. The second column (i.e., a list of intermediate values) contains the list of PIDs with the same state transition history. For example, the first intermediate $<key, value>$ pair indicates that PIDs 0 and 7 have the exact same state transition in both iterations 1 and 2. In iteration 1, the state transition is from SSID 0 to SSID 0. In iteration 2, the state transition is from SSID 1 to SSID 5.

The *reduce*() phase for the second iteration works in a similar manner as for the first.

Map and reduce phases of all subsequent iterations follow the same steps as those in iteration 2. Algorithms 5 and 6 describe in detail the subroutines used in the *map*() and *reduce*() phases of each iterative FEEC computation MapReduce job.

Computation of FEECs using such iterative jobs allow us to identify FEECs without having to compute all possible system state transitions for all permutation of vertices. This approach

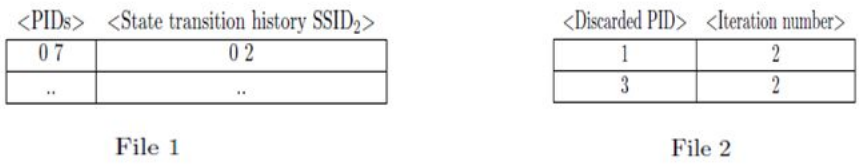


Figure 8.13: Example of output files from second iteration of FEEC computation job. File 1 will be the input file for iteration 3. It shows that PIDs 0 and 7 each produce the state transitions from SSID 0 to SSID 0, and from SSID 1 to SSID 2. Thus, it has not yet been resolved as to whether PIDs 0 and 7 are functionally equivalent. File 2 consists of PIDs that are discarded in iteration 2. These discarded PIDs will not be processed in iteration 3 because PIDs 1 and 3 have been resolved to produce distinct phase spaces.

is most beneficial for GDS maps that produce many different system state transitions for its update sequences so that differences in maps can be identified quickly.

8.2.2.2 Time and Space Complexity

Let n be the number of vertices in the dependency graph X, n_e be the number of edges in the dependency graph X, $|K|$ be the number of vertex states of a single vertex of a GDS. Let $n_{permGrps}$ be the number of groups of permutations of vertices that have the exact same state transitions until iteration $(i-1)$ and n_p be the average number of permutations in each such group. Let $n_{permGrps}{}'$ be the number of groups of permutations of vertices that have the exact same state transitions until iteration i and n'_p be the average number of permutations in each such group. Let n_I be the total number of iterations for which the iterative FEEC computation MapReduce jobs run. Let M and R be the largest number of map and reduce tasks, respectively, that run concurrently in any iteration.

Proposition 5 *The time complexity for the map phases of all the iterative FEEC computation MapReduce jobs is $O(n! \cdot n^2 \cdot |K|^n/M)$. The time complexity for the reduce phases of all the iterative FEEC computation MapReduce jobs is $O(n! \cdot |K|^n/R)$.*

Proof. First we determine the time complexity for each *map*() function (Algorithm 5) that is invoked in the ith iteration of the iterative FEEC computation jobs. The array *pidList*[] in each *map*() function contains n_p PIDs in it. We iterate over all these n_p PIDs to generate the intermediate key-value pairs $<stateTransHistory, PID>$. In each iteration, subroutines $getPerm()$, $getSystemState()$, $computeNextSystemState()$ and $genSSID()$ are used, which have time complexities of $O(n^2)$, $O(n)$, $O(n_e)$ and $O(n)$ respectively. Since $n_e = O(n^2)$ in the worst case, the time complexity for each *map*() function is $O(n_p \cdot n^2)$. This *map*() function is called $n_{permGrps}$ times in the ith iteration of the iterative jobs. Since atmost M map tasks execute concurrently in an iteration, the time complexity for the map tasks of the ith iteration is $O(n_p \cdot n^2 \cdot n_{permGrps}/M)$. Therefore, the time complexity of all the map

Algorithm 5: Algorithm for map phase of each iteration of iterative FEEC computation MapReduce jobs. Section 8.2.2.1 discusses this algorithm in detail.

input : A $<Key, Value>$ pair, where a line of the input file forms the $Value$ and its offset forms the Key.

output: An intermediate $<Key, Value>$ pair denoted as $<stateTransHistory, PID>$, GDS $\mathcal{S}(X, F, \mathcal{W}, K)$ and current iteration i.

Instantiate: Set empty string *pids* and $stateTransHistory$;

//Ignore the Key and process the $Value$, where $Value$ is a line of the input file.

//Each line of the input file consists of the following fields:$<PIDString>$ $<stateTransHistory>$ $<fromSSID>$. See Figure 8.11.

//$<PIDString>$ is a string of PIDs that have the exact same state transitions through iteration $i-1$. $<stateTransHistory>$ is the state transition history and denotes all state transitions computed through iteration $i-1$ for a given $<PIDString>$. $<fromSSID>$ is the system state index that will be processed in the current iteration. For the first iteration, $<PIDString>$ is a single PID whose value lies between 0 and $(n!-1)$, $<stateTransHistory>$ is null and $<fromSSID>$ is 0.

//Extract $PIDString$, state transition history $stateTransHistory$ and SSID $fromSSID$ that will be processed in the current iteration.

$(PIDString, stateTransHistory, fromSSID) = Value.split()$;

//Get each PID from the string $PIDString$.

Array pidList[] = $PIDString.split()$;

//For each PID in $pidList[]$, we consider the GDS to have an initial state $fromSSID$ and then compute its next state.

for $(i = 0; i < pidList.length; i++)$ **do**

  $PID = pidList[i]$;

  //Generate actual permutation of vertices and store it in array *perm*

  $Array\ perm[1..n] = getPerm(PID)$;

  //Generates the state of each vertex of the GDS and stores it in array *sysState*

  $Array\ sysState[1..n] = getSystemState(fromSSID)$;

  //Compute next system state that GDS transitions to.

  $Array\ toSysState[1..n] = computeNextSystemState(perm, sysState, \mathcal{W}, F, X)$;

  //Generate the system state index $toSSID$ that uniquely identifies the system state array $toSysState$.

  $toSysState = genSSID(toSysState)$;

  $emit(stateTransHistory + "" + toSysState, PID)$;

return ;

Algorithm 6: Algorithm for reduce phase of iterative FEEC computation MapReduce jobs. Section 8.2.2.1 discusses this algorithm in detail.

input : An intermediate $<key, value>$ pair, where *key* is a compact representation of a set of state transitions. *value* is a list of PIDs. (This list is denoted by *pidList* in this algorithm)).

output: A $<Key, Value>$ pair where Key is a string of PIDs and $Value$ consists of two fields representing the state transition history and the SSID to be processed in the next iteration.

```
1  Instantiate: Set empty string pidString, count =0, empty string equivClass ;
2  Read the value of SSID to be processed in next iteration and store it in nextSSID ;
3  foreach (pid in pidList) do
4  |   pidString = pidString +" "+pid ;
5  |   count++ ;
6  |   if (count == 1) then
7  |   |   equivClass = pidString ;
8  if (count > 1) then
9  |   Increment the value of global counter numMultiPermKeys by 1 ;
10 |   emit(pidString, equivClass + "" + nextSSID) ;
11 else
12 |   Create a file on HDFS and write (pidString,nextSSID − 1) to it ;
13 return ;
```

tasks over all the n_I sequential iterations of the iterative FEEC computation MapReduce jobs is $O(n_p \cdot n^2 \cdot n_{permGrps} \cdot n_I/M)$. Note that, $n_p \cdot n_{permGrps} = O(n!)$ and in the worst case, $n_I = O(|K|^n)$.

Now we determine the time complexity for each *reduce*() function (Algorithm 6) invoked in the ith iteration of the iterative FEEC computation MapReduce jobs. The list *pidList* in each *reduce*() function has $O(n'_p)$ PIDs in it. We iterate over all PIDs in *pidList* to obtain *pidString* in $O(n'_p)$ time. Thus the time complexity for each *reduce*() function is $O(n'_p)$. This *reduce*() function is called $n'_{permGrps}$ times in the reduce phase of each iteration. Since atmost R reduce tasks execute concurrently in an iteration, the time complexity of the reduce tasks in the ith iteration is $O(n'_p \cdot n'_{permGrps}/R)$ and that of all the reduce tasks over all the n_I iterations is $O(n'_p \cdot n'_{permGrps} \cdot n_I/R)$. Note that, $n'_p \cdot n'_{permGrps} = O(n!)$ and in the worst case, $n_I = O(|K|^n)$. ■

Proposition 6 *Total space complexity for the map phases of all the iterative FEEC computation MapReduce jobs is* $O(M \cdot (n_p + n) + n! \cdot |K|^n)$. *Total space complexity for the reduce phases of all the iterative FEEC computation MapReduce jobs is* $O(n'_p \cdot R + n! \cdot |K|^n)$.

Proof. First we determine the space complexity for each *map*() function (Algorithm 5) called in the ith iteration of the iterative FEEC computation MapReduce jobs. The list *pidList* contains $O(n_p)$ PIDs in it and hence requires $O(n_p)$ space. We require $O(n)$ space to iterate over all PIDs in *pidList* and output key-value pairs $<stateTransHistory, PID>$. Hence each *map*() function requires $O(n_p + n)$ space. This *map*() function is called $n_{permGrps}$ times which produces $n_{permGrps} \cdot n_p$ key-value pairs, each of which requires constant space. An upper bound on the total number of key-value pairs produced by all map tasks over all n_I iterations is $O(n_{permGrps} \cdot n_p \cdot n_I)$, which is $O(n! \cdot n_I)$. With atmost M map tasks being executed concurrently in an iteration, the total space complexity for the map phases of all n_I iterations is $O(M \cdot (n_p + n) + n! \cdot n_I)$ space. In the worst case, $n_I = O(|K|^n)$.

Now we determine the space complexity for each *reduce*() function (Algorithm 6) called in the ith iteration of the iterative FEEC computation jobs. The space complexity for each *reduce*() function is $O(n'_p)$, since both *pidString* and *pidList* require $O(n'_p)$ space. $O(n'_{permGrps})$ numbers of key-value pairs are generated by the reduce tasks of the ith iteration. An upper bound on the total number of key-value pairs generated by all the reduce tasks over all the n_I iterations is $O(n'_{permGrps} \cdot n_I)$. Each of these key-value pairs requires $O(n'_p)$ space. With R reduce tasks being run concurrently, the total space complexity for all reduce tasks over all iterations is $O(n'_p \cdot R + n'_{permGrps} \cdot n_I \cdot n'_p)$. Note that, $n'_p \cdot n'_{permGrps} = O(n!)$ and in the worst case, $n_I = O(|K|^n)$. ■

8.2.2.3 Phase Space Computation Job

This job generates the phase space for all functional equivalence classes of the given GDS. The input to this job comes from two sources:

1. Files containing permutations that were *discarded* by the iterative FEEC computation MapReduce jobs. Each of these *discarded* permutations belong to a separate functional equivalence class.

2. Output from the reduce phase of the last iteration of the iterative FEEC computation MapReduce jobs. This output reflects those functional equivalence classes which have multiple permutations in them.

The *map*() function is implemented as per algorithm 27. The input to a *map*() function is a set of PIDs which belong to the same functional equivalence class. The *map*() function extracts a single PID from this set and computes all system state transitions corresponding to the extracted PID. This forms the phase space of the extracted PID and of all other PIDs belonging to the same functional equivalence class. The phase space of a functional equivalence class and the equivalence class itself are output as a key-value pair. Each key-value pair output by a *map*() function is written to a file by the *reduce*() function.

8.2.2.4 CEEC Computation Job

Computation of cycle equivalence classes is done following the same algorithm as used by the CEEC computation job (as explained in Section 8.2.1.3) of non-iterative brute force GDS evaluation paradigm.

8.3 Vertex Function Agnostic GDS Evaluation

8.3.1 Non-iterative Approach

We implement VFA GDS evaluation on Hadoop MapReduce framework by chaining four MapReduce jobs, as shown in Figure 8.14. Algorithm 28 describes the overall workflow for this evaluation approach.

The chained MapReduce jobs are discussed in detail as follows.

8.3.1.1 VFA FEEC Computation Job

This job uses different VFA schemes (as discussed in Sections 2.5.3 and 2.5.4) to compute functional equivalence classes. Depending on the value of a user specifed input, an appropriate VFA scheme is used for this job. The input to this job is a text file with a unique PID on each line. Hence the input file has $n!$ lines for a sequential GDS. The job outputs $<Key, Value>$ pairs, each of which consists of a set of functionally equivalent PIDs as the

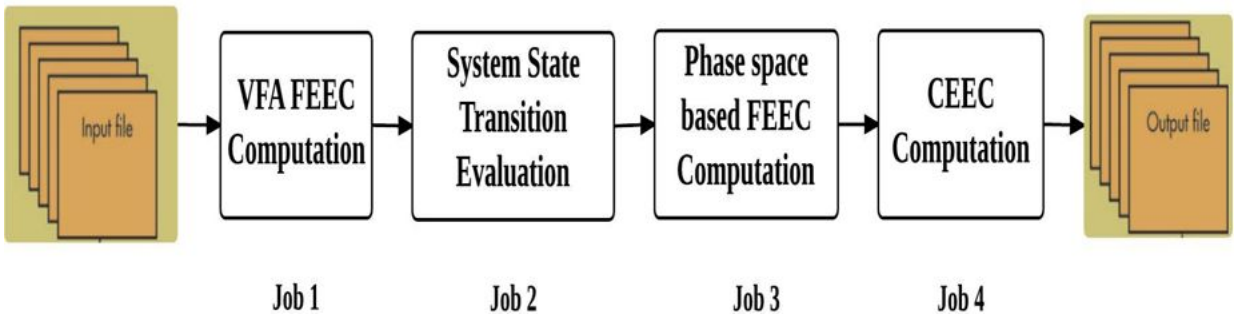


Figure 8.14: Chaining of multiple MapReduce jobs to perform VFA GDS evaluation.

Key and a GDS system state index as the $Value$. MapReduce implementations of various VFA schemes used by this job are discussed as follows.

Scheme 1: Update graph based VFA FEEC computation, with all PIDs being processed.

This scheme is based on the concept of update graph-based VFA FEEC computation discussed in Section 2.5.3. Algorithms 29 and 30 contain the subroutine for the *map*() and *reduce*() functions of this scheme.

The *map*() function takes a key-value pair as its input, with the key being the offset of a line of the input file and the value being the line itself. It processes the line passed to it and extracts the PID from it. It then calls a method $genFEPerms\_UpdateGraph()$ (explained in Algorithm 1) on the extracted PID to generate all permutations that are functionally equivalent to it. The functionally equivalent PIDs are represented using a single string $vfaFEECPerms$. The *map*() function then outputs $<vfaFEECPerms, PID>$ as key-value pairs. Each *reduce*() function receives a unique $vfaFEECPerms$ string output by the *map*() function. It emits $numTotalSysStates$ numbers of key-value pairs for each $vfaFEECPerm$ string that it receives. ($numTotalSysStates$ is the total number of GDS system states.)

Scheme 2: Update Graph based VFA FEEC computation, with selective PIDs being processed.

The input file to the VFA FEEC computation job has all PIDs ordered by their indices.

The *map*() function for this scheme differs from that of the previous scheme in the following manner. The $genFEPerms\_UpdateGraph()$ method called on a parent PID processes this PID and generates all functionally equivalent PIDs from it only if all functionally equivalent PIDs have an index greater than that of the parent PID. If, for a given parent PID, a valid

PID of lesser value is encountered, then procesing terminates and no key-value pair is emitted by the *map*() function. The *reduce*() function for this scheme is the same as that of the previous scheme. Algorithms 31 and 30 describe the *map*() and *reduce*() functions of a VFA FEEC computation job, following Scheme 2. Algorithm 31 uses Algorithm 32.

This scheme offers the following advantages compared to the previous scheme :

1. This scheme reduces the number of redundant computations for the same PID.
2. It requires lesser memory as compared to the previous scheme.
3. The number of key-value pairs produced by the map() task of this scheme is also less compared to that in the previous scheme. This reduces the overhead incurred while shuffling and sorting map output key-value pairs.

The overall workflow of Schemes 1 and 2 are shown in Figure 8.15.

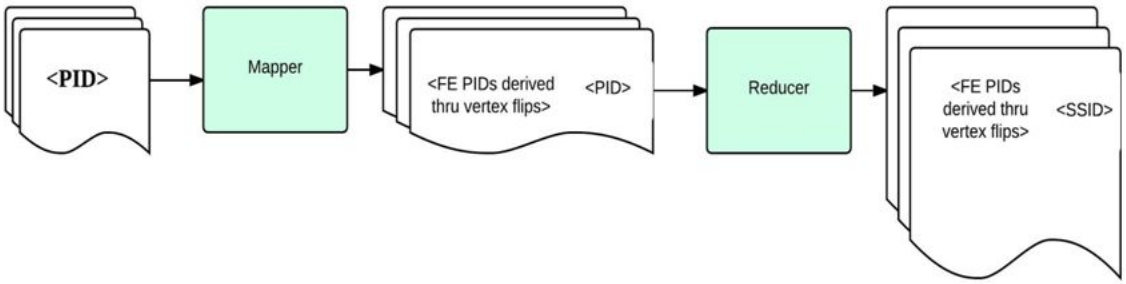


Figure 8.15: VFA FEEC computation job (using schemes 1 and 2) workflow.

Scheme 3: Acyclic orientation based VFA FEEC computation

This scheme is based on the concept of generation of FEECs using acyclic orientations, as discussed in Section 2.5.4. The *map*() and *reduce* functions for this scheme are implemented as per algorithms 33 and 34 respectively.

8.3.1.2 System State Transition Evaluation Job

This job is similar to the system state transition evaluation job used in brute force GDS evaluation (discussed in Section 8.2.1). The difference lies in the format of input to this job. Unlike in brute force GDS evaluation, where the input file consists of $<PID, SSID>$ pairs only, here the input file consists of $<vfaFEECPerms, SSID>$ pairs. $vfaFEECPerms$ is an output from the VFA FEEC computation MapReduce job and is a string obtained by concatenating functionally equivalent PIDs. The *map*() function extracts a single permutation index from this string and processes it as explained in Algorithm 35.

8.3.1.3 Phase Space-based FEEC Computation Job

This job investigates if functional equivalence classes obtained using VFA approach can be merged to form a single equivalence class. This merge is possible only if the phase space of different FEECs are exactly same. The job is similar to the FEEC Computation job (refer Section 8.2.1.2) used in non-iterative brute force GDS evaluation. The only difference lies in the input and output to this job. The input/output value to/from this job is a string of PIDs instead of a single PID as in non-iterative brute force GDS evaluation paradigm.

8.3.1.4 CEEC Computation Job

This job is exactly same as the CEEC computation job of brute force GDS evaluation, discussed in Section 8.2.1.3.

8.3.2 System State-based Iterative Approach

This is an improvement to the brute force system state-based iterative approach (described in Section 8.2.2). We use a VFA FEEC computation job prior to the iterative FEEC computation jobs. The VFA FEEC computation job uses a suitable VFA technique to compute functional equivalence classes. We refer to these FEECs as VFA FEECs. Each line of input to the first iterative job consists of a set of functionally equivalent PIDs and $SSID_0$ (which is 0 as per our convention). Thus the iterative jobs do not process all possible permutation of vertices. Instead they process a single permutation from each VFA FEEC and determine in each iteration i if there are mulitple VFA FEECs that have the exact same state transition in all of the i iterations. We break from an iteration i if the above condition is not satisfied. This GDS evaluation paradigm thus enables us to determine if multiple VFA FEECs do or do not belong to the same functional equivalence class, without having to compute the entire phase space for each VFA FEEC. Figure 8.16 gives the sequence in which different jobs are chained in this paradigm for GDS evaluation. Phase space computation and CEEC computation jobs are implemented following same algorithms (refer Sections 8.2.2.3 and 8.2.2.4) as used in brute force system state-based iterative GDS evaluation.

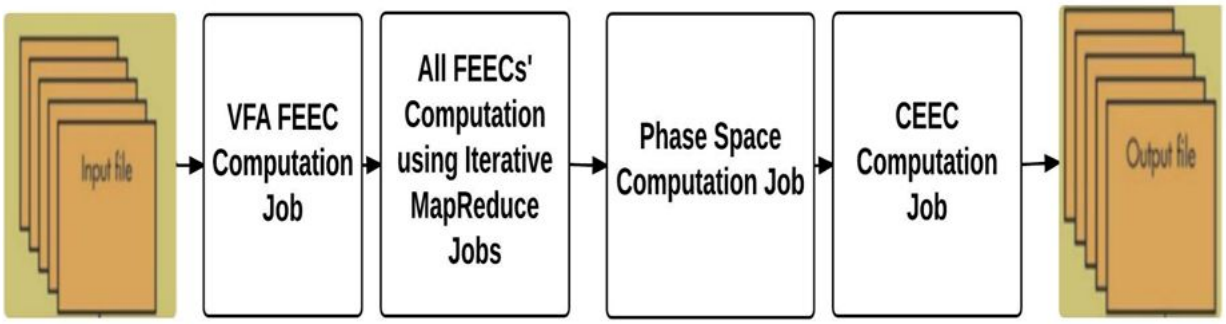


Figure 8.16: VFA GDS evaluation using system state-based iterative approach.

Chapter 9

Performance Evaluation

This chapter presents an evaluation of the different programming paradigms that we have used to carry out GDS analysis. We conduct various experiments to analyze the suitability of different paradigms in modeling various GDSs. We use different plots to show the results of these experiments. All legends used in these plots are abbreviated. Table 9.1 describes what these abbreviations stand for. We use the term **GDS evaluation time** to denote the total time taken to compute all phase spaces, all FEECs and all CEECs of a GDS.

Table 9.1: List of legends used in plots. "Iter" means interative approach; "Non-iter" means non-iterative approach; "BF" is brute force; and "MR" is Map Reduce.

Legends	**Description**
Non-iter BF MR Non-iter BF.	Non-iterative brute force GDS evaluation paradigm using MapReduce.
Non-iter VFA (Acycl) MR, Non-iter VFA (Acycl).	Non-iterative VFA. (with acyclic orientations-based technique) GDS evaluation paradigm using MapReduce.
Non-iter VFA (Update Graph) MR, Non-iter VFA (Update Graph).	Non-iterative VFA (with update graph-based technique) GDS evaluation paradigm using MapReduce.
Iter BF MR, Iter BF.	Iterative brute force GDS evaluation paradigm using MapReduce.
Iter VFA MR, Iter VFA.	Iterative VFA. (with acyclic orientations-based technique) GDS evaluation paradigm using MapReduce.
Serial BF, BF_Serial.	Serial brute force GDS evaluation paradigm
Serial VFA: Update Graph, VFA_Serial_UpdateGraph.	Serial VFA GDS evaluation paradigm, using update graph-based VFA technique.

Serial VFA: Acycl, VFA_Serial_Acyclic, Serial VFA(Acycl).	Serial VFA GDS evaluation paradigm, using acyclic orientations-based VFA technique.

9.1 Experiment Setting

In this section we describe the configuration of the clusters that we use to implement different GDS evaluation paradigms. We also discuss the different configurations of GDS that we use to evaluate different programming paradigms.

9.1.1 Serial Code Implementation Environment

We use TAOS, a large shared memory server, to implement all serial GDS evaluation paradigms. Table 9.2 gives a summary of the specifications of this cluster.

Table 9.2: Specifications for TAOS.

Serial code implementation environment
Large shared memory server. Number of nodes: 1. Processors per node: Four 4-core Xeon E73302, 40GHz. Total cores: 16. Total RAM: 192 GB. Total storage: 300 GB internal, plus direct attached FC. OS: SUSE LINUX Enterprise Server 11.

9.1.2 Hadoop Map Reduce Framework Configuration

We use Cloudera Virtual machine as the development environment for all MapReduce-based GDS evaluation paradigms. This VM runs Hadoop in pseudo-distributed mode. The detailed specification for this VM can be found in Table 9.3.

We use a Hadoop cluster to deploy all MapReduce based GDS evaluation implementations, after successful testing on the Cloudera VM. Table 9.4 summarizes the configuration of this Hadoop Cluster.

Table 9.3: Cloudera VM configuration.

Cloudera QuickStart VM
Large shared memory server.
Number of nodes: 1.
Processors per node: Four 4-core Xeon E73302.40GHz.
Total cores: 16.
Total RAM: 192 GB.
Total storage: 300 GB internal, plus direct attached FC.
OS: SUSE LINUX Enterprise Server 11.

Table 9.4: Hadoop MapReduce cluster configuration.

Version: CDH 5.2x.
VMware Player version: 4.0.6, build-1035888.
Host OS: Windows 8.
Host system RAM: 8GB.
IDE: Eclipse.
Programming language: JAVA.

9.1.3 GDS Configuration

We use different GDS configurations to carry out our experiments. We use different dependency graphs—circle, tree and path, for our experiments. Since all of these graphs generate similar results, we report the results for the circle graph only. Thus all experiments reported in this work, use a Boolean GDS $\mathcal{S}(X, F, \mathcal{W}, K)$ that has a Circle$_n$ graph as its dependency graph. We assign a bi-threshold vertex function to each vertex, where the up threshold and down threshold values are 1 and 3 respectively. (See Section 2.4.) We use both sequential and synchronous update schemes. We vary the value of n, the number of vertices in X, between 4 and 10. Table 9.5 summarizes the different GDS configurations that we are able to study for each GDS evaluation paradigm.

9.2 Results

All experiments and results discussed in Section 9.2.1 pertain to sequential GDSs; while those discussed in Section 9.2.2 are for synchronous GDSs.

Table 9.5: Summary of GDS configurations studied.

GDS evaluation paradigm	**Update Scheme**	**Sizes of dependency graphs**	**Dependency graphs**
Serial brute force	Sequential	4-9	Circle, tree, path.
Serial VFA	Sequential	4-8	Circle, tree, path.
Serial brute force	Synchronous	4-20	Circle, tree, path.
Non-iterative brute force using MapReduce	Sequential	4-9	Circle, tree, path.
Non-iterative brute force using MapReduce	Synchronous	4-18	Circle, tree, path.
Iterative brute force using MapReduce	Sequential	4-9	Circle, tree, path.
Non-iterative VFA using MapReduce and update graph-based VFA technique	Sequential	4-9	Circle, tree, path.
Non-iterative VFA using MapReduce and acyclic orientations-based VFA technique	Sequential	4-10	Circle, tree, path.
Iterative VFA using MapReduce and acyclic orientations-based VFA technique	Sequential	4-10	Circle, tree, path.

9.2.1 Sequential GDS

Figure 9.1 exhibits how the GDS evaluation time using different programming paradigms varies with increasing size of the dependency graph X. Since it is difficult to differentiate between the execution times for smaller graphs, we have blown up this plot and generated another plot (Figure 9.2) which shows the variation in execution time using all GDS evaluation paradigms for circle graphs of size $n \leq 7$. As seen in Figure 9.1, GDS evaluation when done with MapReduce using the iterative brute force technique (Iter BF MR) takes the maximum time amongst all other approaches. This is because of the iterative jobs that are used in this technique. For each Circle_n graph, we run 2^n iterations to compute the FEECs. The size of the input in each iteration is $O(n!)$. Typically the input size decreases over successive iterations. This affects the degree of concurrency that we can achieve in each iteration. This also increases the overhead of the map/reduce jobs compared to the actual computation that is being done in an iteration. The iterative phase of this technique can be thought of as sequentially running 2^n rounds of parallel jobs, where the workload of each of these parallel jobs decreases over successive rounds. For larger graphs (of size $n \geq 8$), the iterative VFA GDS evaluation technique using MapReduce gives us the least execution

time. This is because, unlike in other approaches, this technique enables us to compute the FEECs without having to compute all system state transitions for all possible permutation of vertices. For small graphs (with fewer than 8 nodes), execution time is least when we use serial VFA GDS evaluation paradigms. MapReduce based techniques are not as fast for these small graphs because the workload per map/reduce task is significantly less for smaller graphs and is dwarfed by the overhead required to setup and execute MapReduce jobs. Nonetheless, the execution times ($\leq$ 0.3 hours) are relatively small.

Figure 9.3 shows in detail how the MapReduce GDS evaluation paradigms compare against each other with respect to their execution times. As discussed in Chapter 8, each MapReduce GDS evaluation paradigm uses a chain of MapReduce jobs. The legend of Figure 9.3, from bottom to top, gives the map and reduce operations, in order, for the three MapReduce tasks. Figure 9.3 shows the time taken by each chained component job in each of the five paradigms. Figure 9.4 is the blow up version of the plot seen in Figure 9.3, and it contains results for circle graphs with ≤ 8 vertices. We can make the following observations from the plots shown in Figures 9.3 and 9.4 n:

1. Each evaluation paradigm has one slow job that guides the total execution time for that paradigm. For non-iterative evaluation paradigms, the system state transition evaluation job is the slowest job among all jobs in the chain. This is because the input size of this job is the largest among all jobs that are chained to it. Figures 9.5 and 9.7 illustrate this. For example, in non-iterative brute force GDS evaluation paradigm, it consists of $n!2^n$ key-value pairs; while the input to the FEEC computation and CEEC computation jobs consist of $n!$ and $O(n_{FE})$ (n_{FE} being the number of FEECs) key-value pairs only. Morover the input to this job increases exponentially as we increase the size of the graph. For iterative evaluation paradigms, the iterative jobs consume most of the execution time. Figures 9.6 and 9.8 show that the number of input key-value pairs for the iterative jobs is the largest among all other jobs used in an iterative GDS evaluation paradigm.

2. Total execution time for implementations using a VFA technique is less than that for implementations using brute force technique.

3. Among all VFA schemes, Scheme 3 (discussed in Chapter 8) yields the best performance. This scheme is labeled Iter VFA MR in Figure 9.1. This is because the VFA FEEC computation job, when using this scheme, outputs key-value pairs which are stored as $LongWritable$ objects of the Java MapReduce API. The other schemes use $Text$ objects to store their key-value pairs. $LongWritable$ objects take up less space compared to $Text$ objects, thereby resulting in less disk IO and network IO overheads. Also, shuffling and sorting of intermediate key-value pairs is faster when these are stored as $LongWritable$ objects, compared to when they are stored as $Text$ objects.

To evaluate the performance of the parallel and serial GDS evaluation paradigms we have introduced a term: **Timing ratio**. We define timing ratios T_{serial}, $T_{parallel}$ and $T_{combined}$ as

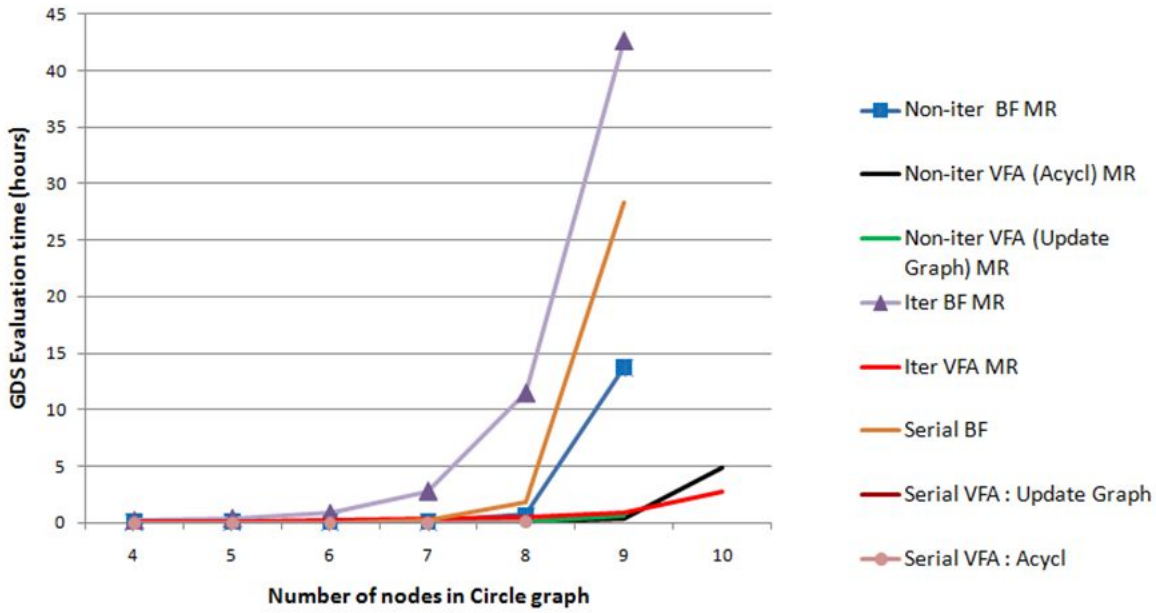


Figure 9.1: Comparison of execution times for all GDS evaluation paradigms. Using serial VFA GDS evaluation paradigms (denoted using legends *Serial VFA: Update Graph* and *Serial VFA: Acycl* in this plot), we are able to perform experiments with Circle graphs having atmost 8 nodes. We are able to perform experiments with Circle graphs having atmost 10 nodes, with the iterative VFA (using acyclic orientations-based VFA technique) MapReduce-based GDS evaluation paradigm (denoted as *Iter VFA MR*) and the non-iterative VFA (using acyclic orientations-based VFA technique) MapReduce-based GDS evaluation paradigm (denoted as *Non-iter VFA (Acycl) MR*). For all the other GDS evaluation paradigms, we are able to perform experiments with Circle graphs having atmost 9 nodes. GDS evaluation time corresponding to the *Non-iter VFA (Acycl) MR* paradigm is lesser than that of the *Non-iter VFA (Update Graph) MR* paradigm, which is lesser than that of the *Iter VFA MR* paradigm, for Circle graphs having lesser than or equal to 9 nodes. GDS evaluation time corresponding to the *Serial VFA: Update Graph* paradigm is lesser than that of the *Non-iter VFA (Update Graph) MR* paradigm for Circle graphs having lesser than or equal to 7 nodes. For a Circle graph with 8 nodes, evaluation time corresponding to the *Serial VFA: Update Graph* paradigm is more than that of the *Non-iter VFA (Update Graph) MR* paradigm. Also, GDS evaluation time using *Non-iter BF MR* paradigm is lesser than that using *Iter VFA MR* paradigm, for Circle graphs having lesser than or equal to 7 nodes.

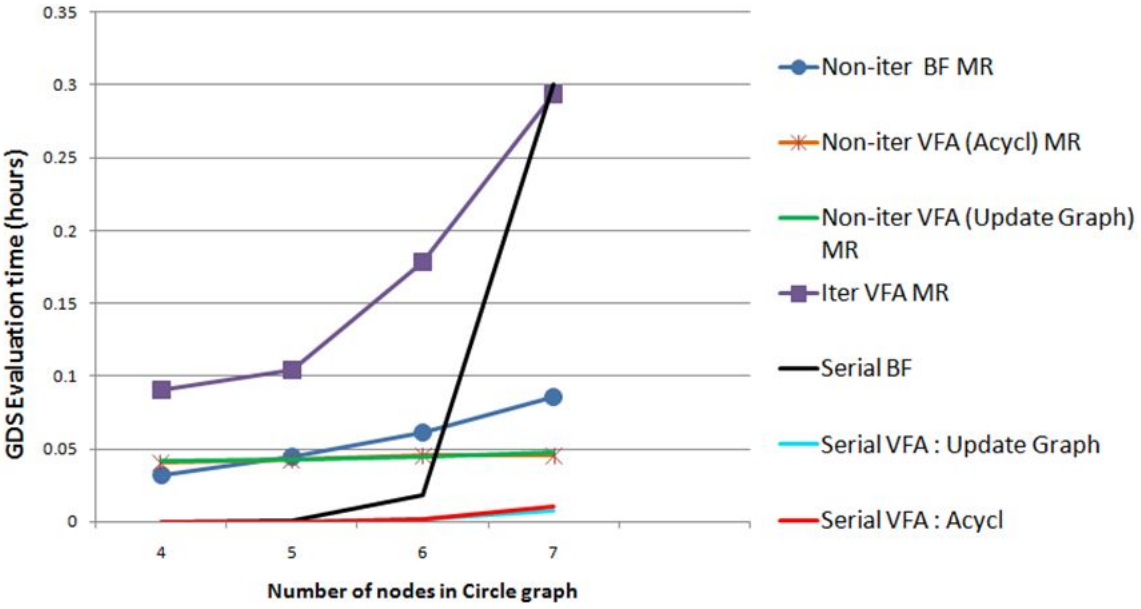


Figure 9.2: Comparison of execution times for all GDS evaluation paradigms, except the iterative brute force MapReduce (Iter BF MR paradigm), using circle graphs with $n \leq 7$ vertices. The execution times for Iter BF MR paradigm is not shown in this plot since they are very high compared to other paradigms.

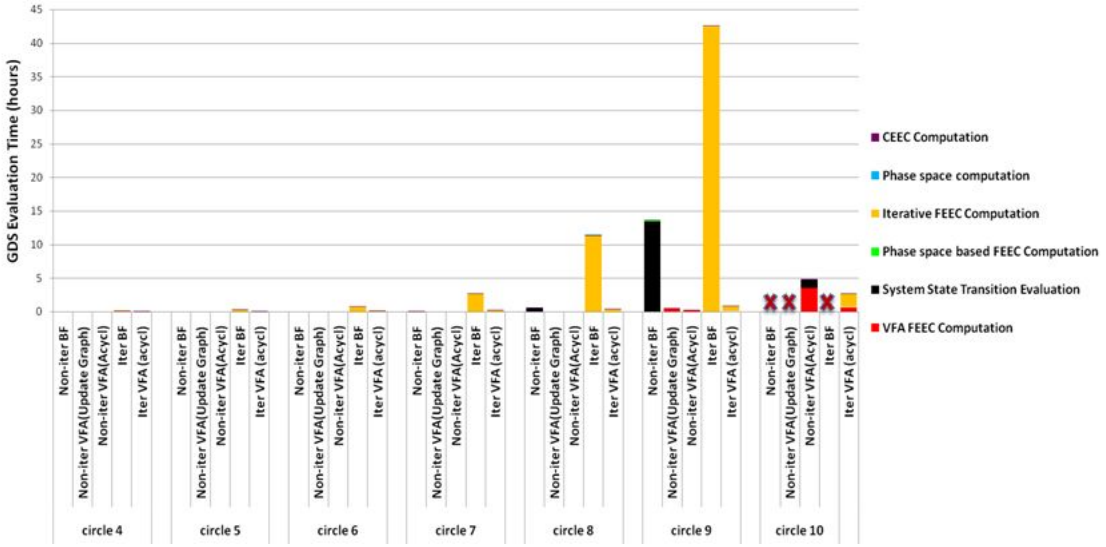


Figure 9.3: Comparison of execution times for all MapReduce GDS evaluation paradigms. The cross marks indicate all those cases which we were unable to run on the given Hadoop cluster owing to memory constraints. The legends on this plot denote different jobs that are used in different GDS evaluation approaches.

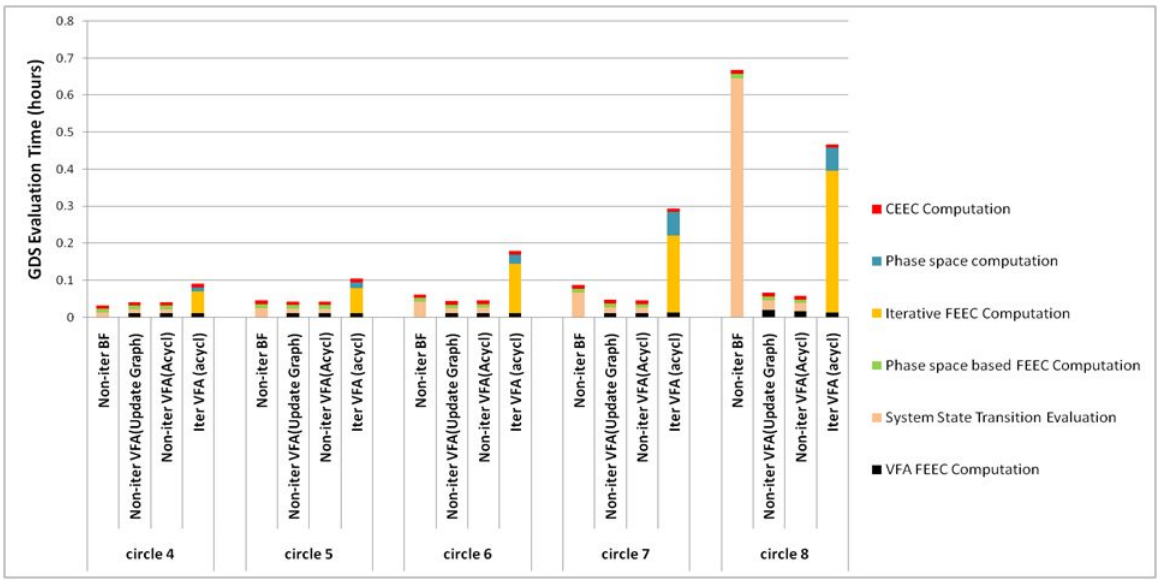


Figure 9.4: Comparison of execution times for all MapReduce GDS evaluation paradigms, using circle graphs with $n \leq 8$ vertices. (This is a blown up version of Figure 9.3.) The legends on this plot denote different jobs that are used in different GDS evaluation approaches.

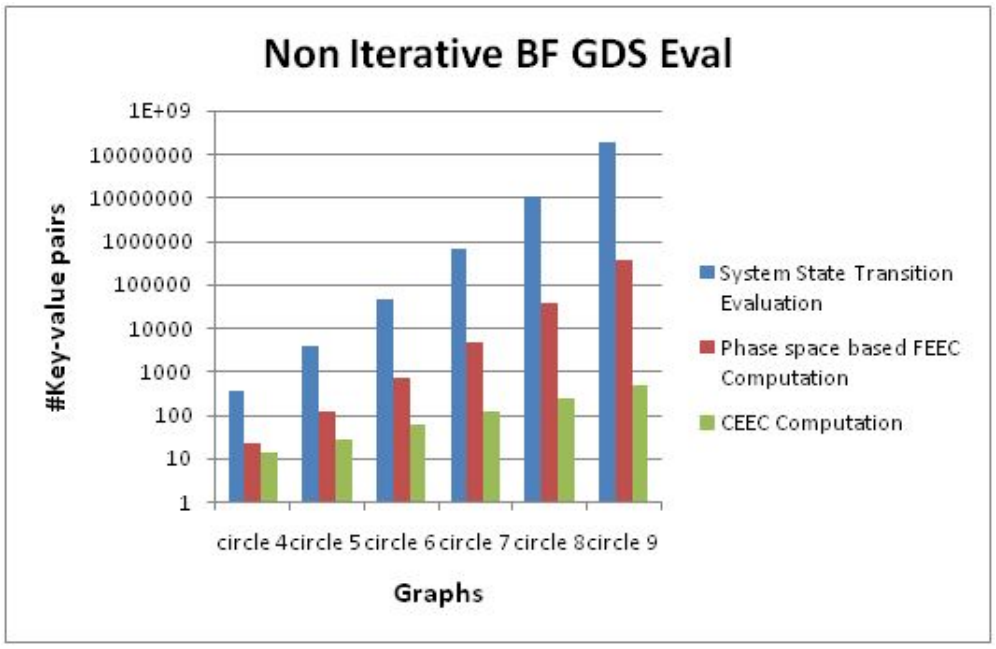


Figure 9.5: Variation in input size for different component jobs of non-iterative brute force GDS evaluation. The legends in this plot denote the component jobs.

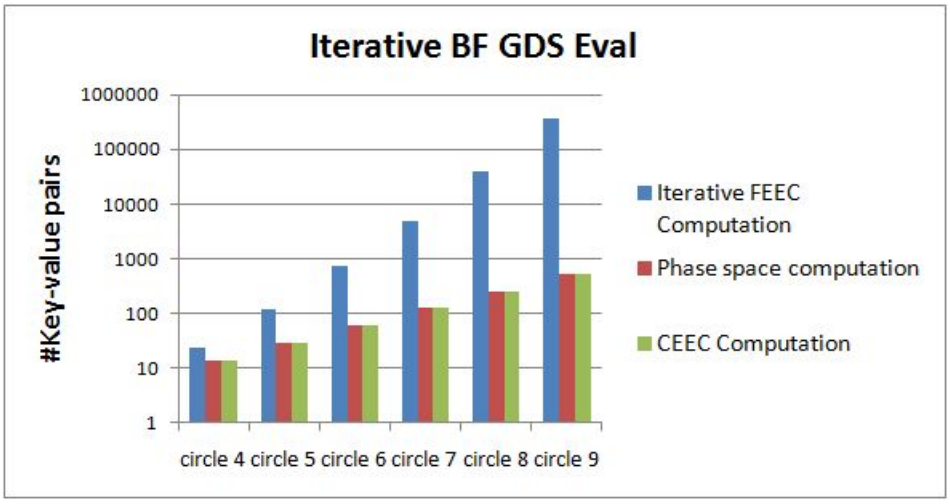


Figure 9.6: Variation in input size for different component jobs of iterative brute force GDS evaluation. The legends in this plot denote the component jobs.

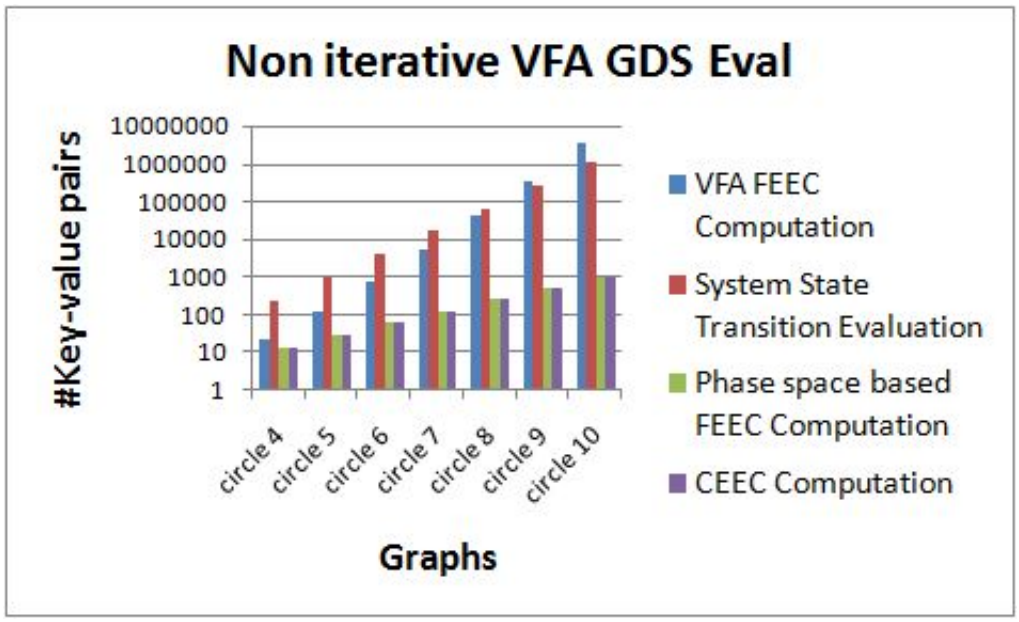


Figure 9.7: Variation in input size for different component jobs of non-iterative VFA GDS evaluation. The legends in this plot denote the component jobs.

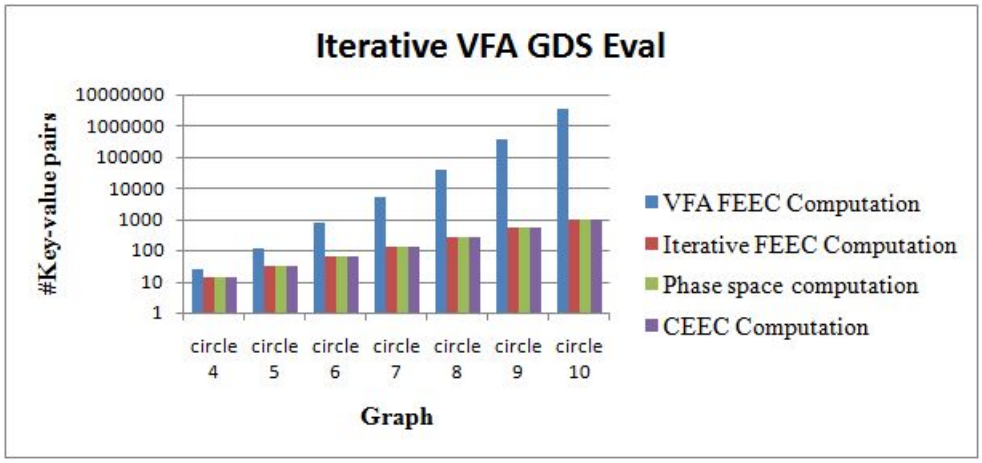


Figure 9.8: Variation in input size for different component jobs of iterative VFA GDS evaluation. The legends in this plot denote the component jobs.

follows. T_{serial} for a serial GDS evaluation paradigm is the ratio of the execution time for the serial implementation of this paradigm to the best execution time obtained with any of our serial evaluation paradigms. $T_{parallel}$ for a parallel GDS evaluation paradigm is the ratio of the execution time for the parallel implementation of this paradigm to the best execution time obtained with any of our parallel GDS evaluation paradigms. $T_{combined}$ is the ratio of the execution time for a serial GDS evaluation paradigm to the best execution time obtained with any of our parallel GDS evaluation paradigms. Figures 9.9, 9.10 and 9.11 present the variations in T_{serial}, $T_{parallel}$ and $T_{combined}$, respectively, for different evaluation techniques that we have used. It is evident from these figures that serial brute force GDS evaluation takes the maximum execution time. One interesting thing to note in the plot for T_{serial} (Figure 9.9) is that the ratio $BF\_Serial/VFA\_Serial\_UpdateGraph$ reaches a peak at $n = 4$ and thereafter dips. This is because the serial brute force evaluation time blows up for smaller n and as n increases, VFA (using acyclic orientations) serial evaluation time blows up, causing the ratio $BF\_Serial/VFA\_Serial\_UpdateGraph$ to come down.

9.2.2 Synchronous GDS

Figure 9.12 compares the variation in GDS evaluation time, for a synchronous GDS, on the serial machine and on the Hadoop MapReduce cluster. GDS evaluation on Hadoop is done using the non-iterative brute force technique. As seen in this plot, GDS evaluation time for small graphs is more on Hadoop than on the serial machine. This is because small graphs result in short MapReduce jobs. The workload per map/reduce task of these short jobs is significantly less. Thus, the MapReduce overhead plays a dominant role in the execution time of such short jobs.

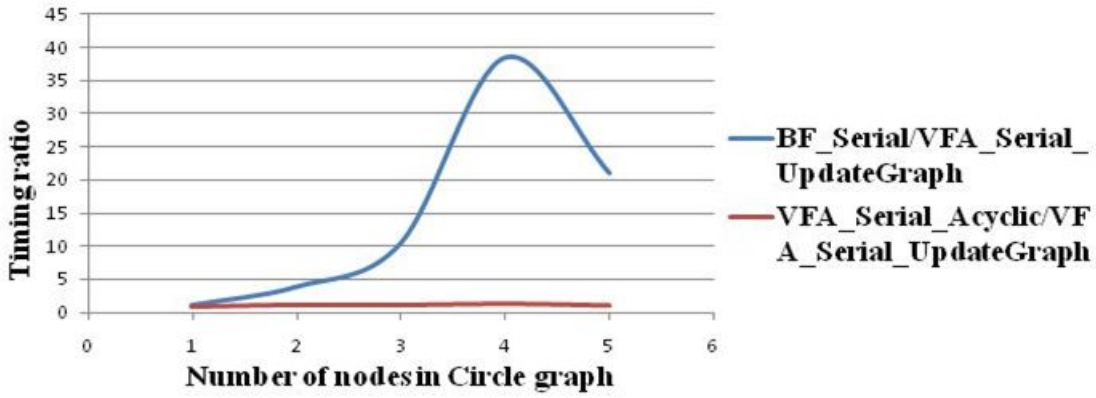


Figure 9.9: Plot showing variation of T_{serial} with size of graph. T_{serial} for a serial GDS evaluation paradigm is the ratio of the execution time for the serial implementation of this paradigm to the best execution time obtained with any of our serial evaluation paradigms.

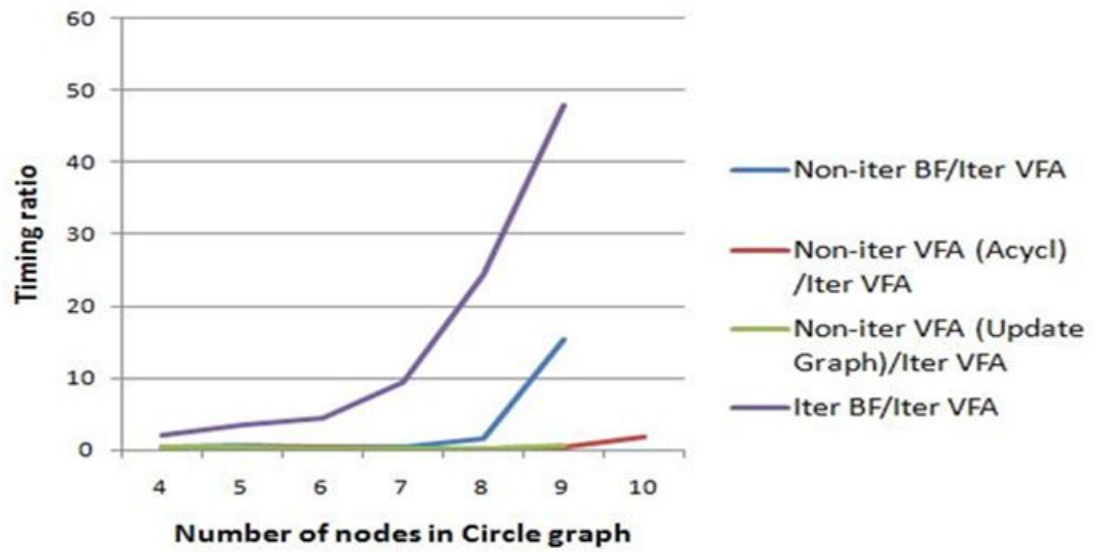


Figure 9.10: Plot showing variation of $T_{parallel}$ with size of graph. $T_{parallel}$ for a parallel GDS evaluation paradigm is the ratio of the execution time for the parallel implementation of this paradigm to the best execution time obtained with any of our parallel GDS evaluation paradigms.

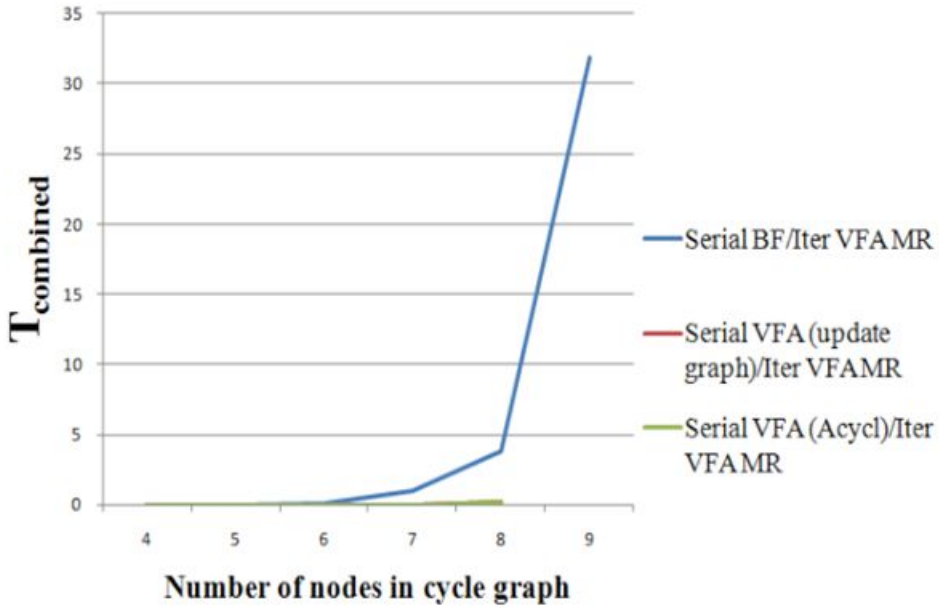


Figure 9.11: Plot showing variation of $T_{combined}$ with size of graph. $T_{combined}$ is the ratio of the execution time for a serial GDS evaluation paradigm to the best execution time obtained with any of our parallel GDS evaluation paradigms.

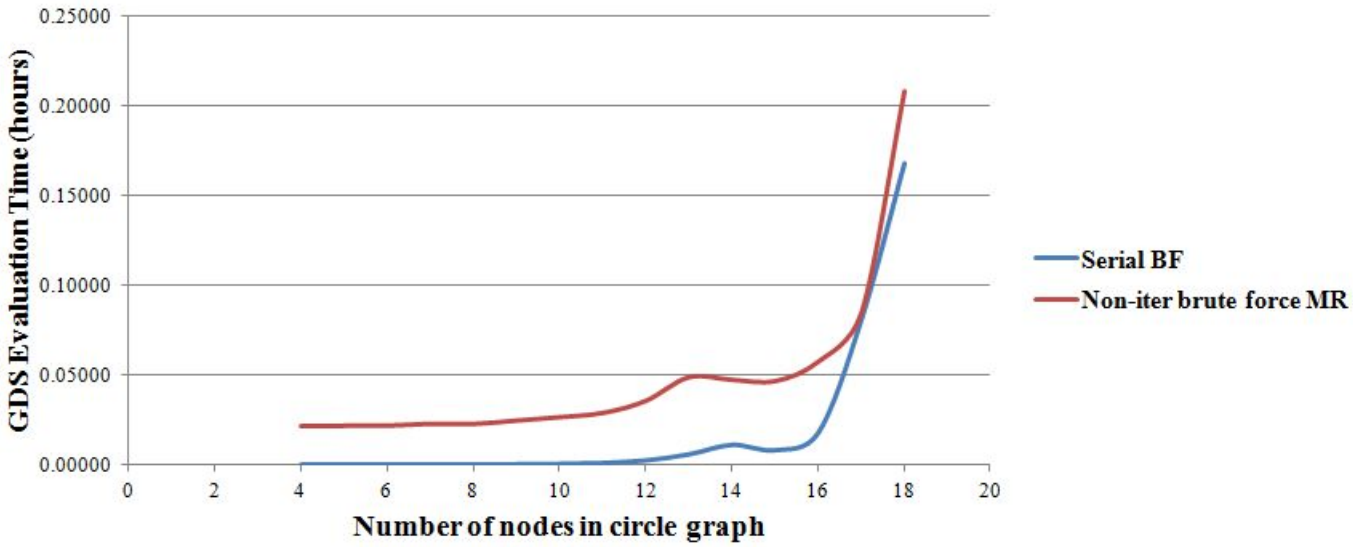


Figure 9.12: Plot comparing evaluation time for synchronous GDS on the serial machine and on the Hadoop MapReduce cluster.

Chapter 10

Conclusion and Future Work

10.1 Conclusion

GDS is a useful mathematical framework to study the dynamics of networked systems. Determining the phase space of a GDS is computationally challenging. In this thesis work, we design various serial and parallel MapReduce algorithms that we use to compute the phase space of a GDS. We borrow theoretical concepts of dynamical systems and use them to design some of these algorithms. The serial algorithms are implemented using C++, while the parallel algorithms are implemented using the Java MapReduce API on Hadoop. We analyze the complexities of each of these algorithms and also perform experiments to evaluate their performances. The serial algorithms work better for smaller graphs, typically with lesser than 8 nodes. For reasons described in Section 1.2, we demonstrate that MapReduce is not suitable in its current form for these types of dynamical systems computations. We also find that we obtain better performance when we use a VFA technique compared to when we do GDS evaluation using brute force approach.

10.2 Future Work

In future we aim to do the following :

1. Use Hadoop Streaming API to perform GDS computations.
2. Model GDS using other parallel programming tools and frameworks, such as Open Science Grid (OSG), MPI, Charm++, GPU, Giraph, Apache Spark, etc.
3. Identify which parallel programming platform is best to model GDS and related systems.

4. Use existing GDS theory to design algorithms for cycle equivalence computation.
5. Building on existing capabilities of GDS Evaluator tool to combine multiple forward trajectories (for limit sets).
6. Explore **delta constant threshold** functions for multistate systems.

Bibliography

[1] *MATLAB Random Boolean Network Toolkit*. MathWorks, Inc., 2007.

[2] *Mathematica*. Wolfram Research, Inc., 2012. Edition 9.0.

[3] *MATLAB and Statistics Toolbox Release 2012b*. MathWorks, Inc., 2012.

[4] Foto N Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D Ullman. Vision paper: Towards an understanding of the limits of Map-Reduce computation. *arXiv preprint arXiv:1204.1754*, 2012.

[5] Maksudul Alam, Maleq Khan, and Madhav V. Marathe. Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing)*, 2013.

[6] Shaikh Arifuzzaman, Maleq Khan, and Madhav V. Marathe. PATRIC: A parallel algorithm for counting triangles in massive networks. In *ACM Conference on Information and Knowledge Management (CIKM)*, 2013.

[7] R Agrawal AS941 and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of International Conference on Very Large Databases (VLDB 94)*, pages 487–499, 1994.

[8] Chris Barrett, Harry B Hunt, Madhav V Marathe, SS Ravi, Daniel J Rosenkrantz, and Richard E Stearns. Modeling and analyzing social network dynamics using stochastic discrete graphical dynamical systems. *Theoretical Computer Science*, 412(30):3932–3946, 2011.

[9] Chris Barrett, Harry B Hunt, Madhav V Marathe, SS Ravi, Daniel J Rosenkrantz, Richard E Stearns, and Mayur Thakur. Predecessor existence problems for finite discrete dynamical systems. *Theoretical Computer Science*, 386(1):3–37, 2007.

[10] Chris Barrett, Harry B Hunt Iii, Madhav V Marathe, SS Ravi, Daniel J Rosenkrantz, and Richard E Stearns. Reachability problems for sequential dynamical systems with threshold functions. *Theoretical Computer Science*, 295(1):41–64, 2003.

[11] Christopher L Barrett, Harry B Hunt, Madhav V Marathe, SS Ravi, Daniel J Rosenkrantz, and Richard E Stearns. Complexity of reachability problems for finite discrete dynamical systems. *Journal of Computer and System Sciences*, 72(8):1317–1345, 2006.

[12] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[13] Chris Bunch, Brian Drawert, and Matthew Norman. MapScale: a cloud environment for scientific computing. *University of California, Computer Science Department, Tech. Rep*, 2009.

[14] Damon Centola and Michael Macy. Complex contagions and the weakness of long ties1. *American Journal of Sociology*, 113(3):702–734, 2007.

[15] Claudine Chaouiya, Aurélien Naldi, and Denis Thieffry. Logical modelling of gene regulatory networks with ginsim. In *Bacterial Molecular Networks*, pages 463–479. Springer, 2012.

[16] Jonathan Cohen. Graph twiddling in a MapReduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[17] Tolga Dalman, Tim Dörnemann, Ernst Juhnke, Michael Weitzel, Wolfgang Wiechert, Katharina Nöh, and Bernd Freisleben. Cloud MapReduce for Monte Carlo bootstrap applied to metabolic flux analysis. *Future Generation Computer Systems*, 29(2):582–590, 2013.

[18] Hidde De Jong. Modeling and simulation of genetic regulatory systems: a literature review. *Journal of computational biology*, 9(1):67–103, 2002.

[19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] Jacques Demongeot, Eric Goles, Michel Morvan, Mathilde Noual, and Sylvain Sené. Attraction basins as gauges of robustness against boundary conditions in biological complex systems. *PloS one*, 5(8):e11793, 2010.

[21] Elena Dubrova. Bns (Boolean networks with synchronous update), 2013.

[22] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[23] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. MapReduce for data intensive scientific analyses. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 277–284. IEEE, 2008.

[24] Nazim Fates. Fiatlux, 2013.

[25] The Apache Software Foundation. Hadoop MapReduce Tutorial. 2013.

[26] Uwe Freiwald and Jörg R Weimar. The Java based cellular automata simulation system–JCASim. *Future Generation Computer Systems*, 18(7):995–1004, 2002.

[27] Abhishek Garg, Alessandro Di Cara, Ioannis Xenarios, Luis Mendoza, and Giovanni De Micheli. Synchronous versus asynchronous modeling of gene regulatory networks. *Bioinformatics*, 24:1917–1925, 2008.

[28] Carlos Gershenson. Classification of random Boolean networks. In *Artificial Life VIII:Proceedings of the Eight International Conference on Artificial Life*, pages 1–8, 2002.

[29] Carlos Gershenson. Introduction to random Boolean networks. In *Workshop and Tutorial Proceedings, Ninth International Conference on the Simulation and Synthesis of Living Systems (Artificial Life IX)*, pages 160–173, 2004. arXiv: http://arxiv.org/pdf/nlin/0408006.pdf.

[30] Carlos Gershenson. *RBNLab*. 2014.

[31] A Gonzalez Gonzalez, Aurélien Naldi, Lucas Sanchez, Denis Thieffry, and Claudine Chaouiya. Ginsim: a software suite for the qualitative modelling, simulation and analysis of regulatory networks. *Biosystems*, 84(2):91–100, 2006.

[32] Mark Granovetter. Threshold models of collective behavior. *American journal of sociology*, pages 1420–1443, 1978.

[33] Tomáš Helikar, Bryan Kowal, Sean McClenathan, Mitchell Bruckner, Thaine Rowley, Alex Madrahimov, Ben Wicks, Manish Shrestha, Kahani Limbu, and Jim A Rogers. The Cell Collective: Toward an open and collaborative approach to systems biology. *BMC Systems Biology*, 6(1):1–14, 2012.

[34] Tomáš Helikar and Jim A Rogers. ChemChains: A platform for simulation and analysis of biochemical networks aimed to laboratory scientists. *BMC Systems Biology*, 3(1):1–15, 2009.

[35] Franziska Hinkelmann, Madison Brandon, Bonny Guang, Rustin McNeill, Grigoriy Blekherman, Alan Veliz-Cuba, and Reinhard Laubenbacher. Adam: analysis of discrete models of biological systems using computer algebra. *BMC bioinformatics*, 12(1):295, 2011.

[36] Tim Hutton, Robert Munafo, Andrew Trevorrow, Tom Rokicki, and Dan Wills. Ready, a cross-platform implementation of various reaction-diffusion systems.

[37] Masakuni Ishii, Jungkyu Han, and Hiroaki Makino. Design and performance evaluation for hadoop clusters on virtualized environment. In *Information Networking (ICOIN), 2013 International Conference on*, pages 244–249. IEEE, 2013.

[38] Pelle Jakovits and Satish Narayana Srirama. Evaluating MapReduce frameworks for iterative scientific computing applications. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 226–233. IEEE, 2014.

[39] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of MapReduce: An in-depth study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.

[40] Stuart A Kauffman. Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of theoretical biology*, 22(3):437–467, 1969.

[41] Myoungjin Kim, Hanku Lee, and Yun Cui. Performance evaluation of image conversion module based on MapReduce for transcoding and transmoding in SMCCSE. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 396–403. IEEE, 2011.

[42] Steffen Klamt, Julio Saez-Rodriguez, and Ernst D Gilles. Structural and functional analysis of cellular networks with CellNetAnalyzer. *BMC Systems Biology*, 1(1):1–2, 2007.

[43] Steffen Klamt, Joerg Stelling, Martin Ginkel, and Ernst Dieter Gilles. FluxAnalyzer: exploring structure, pathways, and flux distributions in metabolic networks on interactive flux maps. *Bioinformatics*, 19(2):261–269, 2003.

[44] Chris J. Kuhlman, V.S. Kumar, Madhav V. Marathe, Henning S. Mortveit, Samarth Swarup, Gaurav Tuli, S. S. Ravi, and Daniel J. Rosenkrantz. A general-purpose graph dynamical system modeling framework. In *Proceedings of the Winter Simulation Conference*, pages 296–308. Winter Simulation Conference, 2011.

[45] Chris J. Kuhlman and Henning S. Mortveit. Attractor stability in nonuniform Boolean networks. *Theoretical Computer Science*, pages 20–33, 2014.

[46] Chris J. Kuhlman and Henning S Mortveit. Limit sets of generalized, multi-threshold networks. *Journal of Cellular Automata*, 10:161–193, 2015.

[47] Chris J. Kuhlman, Henning S. Mortveit, David Murrugarra, and V. S. Anil Kumar. Bifurcations in Boolean networks. *Automata*, pages 29–46, 2011.

[48] Mustafa RS Kulenovic and Orlando Merino. *Discrete Dynamical Systems and Difference Equations with Mathematica*. Chapman-Hall, 2002.

[49] VS Anil Kumar, Matt Macauley, and Henning S Mortveit. Limit set reachability in asynchronous graph dynamical systems. In *Reachability Problems*, pages 217–232. Springer, 2009.

[50] Edward A Lee and Haiyang Zheng. Operational semantics of hybrid systems. In *Hybrid Systems: Computation and Control*, pages 25–53. Springer, 2005.

[51] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. Moon: MapReduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 95–106. ACM, 2010.

[52] Jimmy Lin. MapReduce is good enough? If all you have is a hammer, throw away everything that's not a nail! *arXiv preprint arXiv:1209.2191*, 2012.

[53] Matthew Macauley, Jon McCammond, and Henning S Mortveit. Order independence in asynchronous cellular automata. *arXiv preprint arXiv:0707.2360*, 2007.

[54] Matthew Macauley and Henning S Mortveit. Cycle equivalence of graph dynamical systems. *Nonlinearity*, 22(2):421, 2009.

[55] Matthew Macauley and Henning S Mortveit. Update sequence stability in graph dynamical systems. *Discrete and Continuous Dynamical Systems - Series S*, 4(6):1533–1541, 2011.

[56] Michael Macy. Chains of cooperation: Threshold effects in collective action. *American Sociological Review*, 56:730–747, 1991.

[57] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[58] Henning Mortveit and Christian Reidys. *An introduction to sequential dynamical systems*. Springer Science & Business Media, 2007.

[59] Christoph Müssel, Martin Hopfensitz, and Hans A Kestler. BoolNet–an R package for generation, reconstruction and analysis of Boolean networks. *Bioinformatics*, 26(10):1378–1380, 2010.

[60] Claudius Ptolemaeus. *System Design, Modeling, and Simulation using Ptolemy II*. lulu.com, 2013.

[61] B Thirumala Rao and LSS Reddy. Survey on improved scheduling in Hadoop MapReduce in cloud environments. *arXiv preprint arXiv:1207.0780*, 2012.

[62] John Reid. Pybool: A Python package to infer Boolean networks under constraints, 2011.

[63] Gonzalo Ruz, Eric Goles, et al. Reconstruction and update robustness of the mammalian cell cycle network. In *Computational Intelligence in Bioinformatics and Computational Biology (CIBCB), 2012 IEEE Symposium on*, pages 397–403. IEEE, 2012.

[64] Prashant Sethia and Kamalakar Karlapalem. A multi-agent simulation framework on small hadoop cluster. *Engineering Applications of Artificial Intelligence*, 24(7):1120–1127, 2011.

[65] I. Shmulevich. *PBN*. 2014.

[66] Ilya Shmulevich, Edward R Dougherty, and Wei Zhang. From Boolean to probabilistic Boolean networks as models of genetic regulatory networks. *Proceedings of the IEEE*, 90(11):1778–1792, 2002.

[67] Satish Narayana Srirama, Pelle Jakovits, and Eero Vainikko. Adapting scientific computing problems to clouds using MapReduce. *Future Generation Computer Systems*, 28(1):184–192, 2012.

[68] Brandilyn Stigler. Polynomial dynamical systems in systems biology. In *Proceeding of Symposia in Applied Mathematics*, pages 59–84, 2006.

[69] Gautier Stoll, Eric Viara, Emmanuel Barillot, and Laurence Calzone. Continuous time Boolean modeling for biological signaling: application of Gillespie algorithm. *BMC Systems Biology*, 6(1):116–1–116–18, 2012.

[70] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(1-3):66–73, 2010.

[71] Rene Thomas. Regulatory networks seen as asynchronous automata: A logical description. *Journal Theoretical Biology*, 153:1–23, 1991.

[72] A Trevorrow and T Rokicki. Golly: open source, cross-platform application for exploring Conways Game of Life and other cellular automata, 2009.

[73] Tiankai Tu, Charles A Rendleman, David W Borhani, Ron O Dror, Justin Gullingsrud, MO Jensen, John L Klepeis, Paul Maragakis, Patrick Miller, Kate A Stafford, et al. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[74] ISTV AN VAJK. Performance evaluation of Apriori algorithm on a Hadoop cluster.

[75] Guozhang Wang, Marcos Vaz Salles, Benjamin Sowell, Xun Wang, Tuan Cao, Alan Demers, Johannes Gehrke, and Walker White. Behavioral simulations in MapReduce. *Proceedings of the VLDB Endowment*, 3(1-2):952–963, 2010.

[76] Duncan J Watts. A simple model of global cascades on random networks. *Proceedings of the National Academy of Sciences*, 99(9):5766–5771, 2002.

[77] Katja Wegner, Johannes Knabe, Mark Robinson, Attila Egri-Nagy, Maria Schilstra, and Chrystopher Nehaniv. The Netbuilder project: Development of a tool for constructing, simulating, evolving, and analysing complex regulatory networks. *BMC Systems Biology*, 1(1):1–2, 2007.

[78] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[79] Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601–644, 1983.

[80] Andy Wuensche. *Exploring Discrete Dynamics*. Luniver Press, 2011.

[81] Andy Wuensche. *DDLab*. 2014.

[82] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

Appendix A

List of Symbols

Table A.1: List of symbols used.

Symbol	**Description**
GDS	Graph dynamical system.
$\mathcal{S}$	A graph dynamical system.
X	Dependency graph of a GDS.
n	Number of vertices in X.
v[X]	Vertex set of X.
$d^{in}(v)$	In-degree of a vertex v in X.
n_e	Number of edges in X.
e[X]	Edge set of X.
F	Sequence of vertex functions of a GDS.
f_v	Vertex function of a vertex v in X.
$\mathcal{W}$	Update scheme of a GDS.
K	Vertex state set of a GDS.
x_v	Vertex state of a vertex v in X.
x	A GDS system state.
$n[v]$	Sequence of vertices, sorted in increasing order, in the 1-neighborhood of a vertex v in X.
$x[v]$	Restricted state of a vertex v in X.
$x(t)$	GDS system state at time t.
$x(t)[v]$	Restricted state of vertex v (in X) at time t.
S_X	The set of all permutations of $v[X]$.
π	A permutation of vertices $\in S_X$.
π_i	The ith element in π.
π_B	A block permutation of vertices in X.
F	A GDS map.

F_v	X-local function for a vertex v in X.
$\mathbf{F}_\pi$	A sequential or asynchronous GDS map.
$\mathbf{F}_{\pi_B}$	A block sequential GDS map.
$\Gamma(\mathbf{F})$	Phase space of the GDS map $\mathbf{F}$.
$\omega(x)$	Omega-limit set of a GDS system state x.
k_{01}	Up-threshold value in a bi-threshold vertex function.
k_{10}	Down-threshold value in a bi-threshold vertex function.
$U(X)$	Update graph of X.
$O(X)$	An acyclic orientation of an undirected dependency graph X.
Acyc(X)	Set of all acyclic orientations of an undirected dependency graph X.
BF	Brute force.
VFA	Vertex function agnostic.
FEEC	Functional equivalence class.
VFA FEEC	A FEEC computed using a VFA technique.
CEEC	Cycle equivalence class.
MR	MapReduce.
HDFS	Hadoop distributed file system.
PID	A unique permutation index.
SSID	A unique system state index.
M	Number of concurrent map tasks.
R	Number of concurrent reduce tasks.
n_{FE}	Number of FEECs.
n_{CE}	Number of CEECs.
n_{vfaFE}	Number of FEECs computed using a VFA technique.
n_{fePIDs}	Average number of PIDs in a single FEEC of a GDS.
$n_{vfafePIDs}$	Average number of PIDs in a single VFA FEEC.
n_{cePIDs}	Average number of PIDs in a single CEEC of a GDS.
n_{CS}	The length of the largest cycle among all phase space graphs of a GDS.
$n_{permGrps}$	The number of groups of permutations of vertices that have the exact same state transitions until iteration $(i-1)$ of the iterative FEEC computation MapReduce jobs.
n_p	The average number of permutations in each of the $n_{permGrps}$ groups.
$n'_{permGrps}$	The number of groups of permutations of vertices that have the exact same state transitions until iteration i of the iterative FEEC computation MapReduce jobs.
n'_p	The average number of permutations in each of the $n'_{permGrps}$ groups.
n_I	Total number of iterations of the iterative FEEC computation MapReduce jobs.
CA	Cellular automata.
BN	Boolean network.

RBN	Random Boolean network.
PBN	Probabilistic Boolean network.
PDS	Polynomial dynamical system.

Appendix B

Algorithms

B.1 Serial GDS Evaluation

B.1.1 Serial Algorithms

B.1.2 Time and Space Complexity for Serial Brute Force GDS Evaluation Algorithm

Let n be the number of vertices in the dependency graph X, n_e be the number of edges in X, n_{FE} be the number of FEECs, n_{CE} be the number of CEECs, n_{fePIDs} be the average number of PIDs in a single FEEC, n_{cePIDs} be the average number of PIDs in a single CEEC, n_{CS} be the length of the largest cycle among all phase space graphs of a GDS.

Proposition 7 *The time complexity for the serial brute force GDS evaluation algorithm (Algorithm 8) is* $O(|K|^n \cdot n! \cdot (n^2 + n_{FE}) + n_{FE} \cdot (|K|^n + n_{CE} \cdot n_{CS}))$.

Proof. We assume that the time taken to add and delete elements to and from all map and set data structures used in Algorithms 7, 8, 9 and 10 is constant. Time complexity for the serial brute force GDS evaluation algorithm (Algorithm 8) can be determined as follows. It takes $O(n \cdot n!)$ time to populate $permMap$ and $O(n \cdot |K|^n)$ time to populate $sysStMap$. Algorithm 7 is used to populate $sysStMap$. We iterate over all PIDs in $permMap$ and determine the phase space for each of these PIDs. The phase space for a PID is obtained by iterating over all $|K|^n$ SSIDs in $sysStMap$ and computing that system state which the GDS transitions to, from each SSID. In each iteration, $computeNextSysState()$ takes $O(n_e)$ time to compute the next system state that the GDS transitions to, from a system state $systemState$. Subroutine $convertToSSID()$ takes $O(n)$ time to generate $nextSSID$ from system state array $nextSystemState[]$. Hence generation of the phase space for a PID takes

Algorithm 7: Algorithm to populate system state map. Refer Section 7.1 for definition of this map.

input : Number of vertices n in the GDS and number of states that a vertex of the GDS can be in. The latter is denoted as $radix$ and is given by the size of vertex state set $|K|$.

output: System state map denoted as $sysStMap < SSID, sysState >$.

```
1  Instantiate: empty Arrays prevSysSt[1...n] and nextSysSt[1...n], empty map
   sysStMap < SSID, sysState > ;
2  //Set starting index for a system state.
3  currSSID = 0 ;
4  Set state of all vertices to 0 by assigning 0 to all elements of prevSysSt[1...n] ;
5  //Calculate total number of system states.
6  totSysState = radix^n ;
7  sysStMap.add(currSSID, prevSysSt) ;
8  currSSID++ ;
9  // Compute subsequent SSIDs.
10 for (i = 1; i ≤ totSysState; i++) do
11     Copy contents of prevSysSt to nextSysSt ;
12     for (j = n; j ≥ 1; j--) do
13         nextSysSt[j] = nextSysSt[j] + 1 ;
14         if (nextSysSt[j] == radixx) then
15             if (j ≠ 0) then
16                 nextSysSt[j] = 0 ;
17         else
18             break ;
19     sysStMap.add(currSSID, nextSysSt) ;
20     currSSID++ ;
21     Copy contents of nextSysSt[] to prevSysSt[] ;
22 return ;
```

Algorithm 8: Algorithm used to perform serial brute force GDS evaluation. Section 7.2 discusses this algorithm in detail.

input : A GDS $\mathcal{S}(X, F, \mathcal{W}, K)$.
output: Phase spaces, functional equivalence classes and cycle equivalence classes.

Instantiate: empty map $pidPhaseSpaceMap < PID, PhaseSpace >$, empty map $permMap < PID, perm >$, empty map $sysStMap < SSID, sysState >$, empty map $feecMap < feec, PIDSet >$, empty map $feecPhaseSpaceMap < feec, PhaseSpace >$;

Populate maps $permMap$ and $sysStMap$ (using schemes discussed in Section 7.1 and Algorithm 7) ;
foreach *(entry in permMap)* **do**
    $PID = entry.getPID()$;
    Array $perm[1..n] = entry.getPerm()$;
    //Compute the phase space for PID.
    foreach *(entry in sysStMap)* **do**
        $SSID = entry.getSSID()$;
        Array $systemState[1..n] = entry.getSystemState()$;
        Array $nextSystemState[1..n] =$
        $computeNextSysState(perm, systemState, \mathcal{W}, F, X)$;
        //Generate the SSID that encodes $nextSystemState[]$. Refer to Algorithm 16.
        $nextSSID = convertToSSID(nextSystemState)$;
    Insert PID and the above computed phase space (PS) into $pidPhaseSpaceMap$;
    //Refer Algorithm 9.
    computeFEEC($pidPhaseSpaceMap$, $feecMap$, $feecPhaseSpaceMap$) ;
//Refer Algorithm 10.
$computeCEEC$(feecMap,feecPhaseSpaceMap) ;
return ;

Algorithm 9: Serial algorithm used to identify the FEEC that a PID belongs to. Algorithm 8 uses this algorithm. Section 7.2 describes this algorithm in detail.

input : *pidPhaseSpaceMap*, $feecMap$, $feecPhaseSpaceMap$. (Refer Section 7.1 for the definitions of these maps.)
output: FEEC that a PID belongs to.

```
1  PID = pidPhaseSpaceMap.getPID() ;
2  //Get Phase Space for PID.
3  PS = pidPhaseSpaceMap.getPS() ;
4  Iterate over feecMap to get all keys from this map and store the keys in set feecSet ;
5  //These keys are PIDs representing FEECs.
6  isInExistingClass = false ;
7  foreach (feec in feecSet) do
8      //Each element of feecSet is a PID representing a FEEC.
9      //Get Phase space corresponding to this FEEC.
10     feecPS = feecPhaseSpaceMap.getPS(feec) ;
11     //Equality in the next IF statement means that PS and feecPS have the same
       state transitions.
12     if (PS == feecPS) then
13         //PID belongs to an existing equivalence class. Hence add PID to the existing
           equivalence class.
14         feecMap.addPID(feec, PID) ;
15         isInExistingClass = true ;
16         break ;
17 if (isInExistingClass == false) then
18     //PID does not belong to an existing FEEC. Hence create a new equivalence class
       and add PID to that class.
19     feecMap.insertEC(PID, PID) ;
20     Insert < PID, PS > into ecPhaseSpaceMap ;
21 //Clear all entries from the map.
22 pidPhaseSpaceMap.clearAll() ;
23 return ;
```

Algorithm 10: Serial algorithm to compute CEECs. Algorithms 8 and 11 use this algorithm. Section 7.2 describes this algorithm in detail.

input : $feecMap$ and $feecPhaseSpaceMap$. (Refer to Section 7.1 for definiton of these maps.)
output: $ceecMap$, $PIDCycleStructMap$.

```
1  Instantiate: empty maps ceecMap and ceecMap ;
2  Iterate over feecMap to get all keys from this map and store the keys in set feecSet
   //These keys are PIDs representing FEECs.
3  isInExistingClass = false ;
4  foreach (feec in feecSet) do
5      //Each element of feecSet is a PID representing a FEEC. We get the phase space
       corresponding to this PID.
6      feecPS = feecPhaseSpaceMap.getPS(feec) ;
7      Use feecPS to determine the strongly connected components (SCCs) of the phase
       space graph for feec. The SCCs are the limit cycles, called csc ;
8      foreach (entry in ceecMap) do
9          ceec = entry.getCeec() ;
10         //Get cycle structure for ceec.
11         ceecCsc = PIDCycleStructMap.getCsc(ceec) ;
12         //Equality in the next IF statement means that ceecCsc and csc have the
           same limit cycle sizes and the same number of cycles of each size.
13         if (ceecCsc == csc) then
14             //feec belongs to an existing CEEC.
15             ceecMap.add(ceec, feec) ;
16             isInExistingClass = true ;
17             break ;
18     if (isInExistingClass == false) then
19         //Create a new CEEC and add feec to it.
20         ceecMap.insert(feec, feec) ;
21         PIDCycleStructMap.insert(feec, csc) ;
22 return ;
```

Algorithm 11: Algorithm used to perform serial VFA GDS evaluation. Section 7.3 describes this algorithm in detail.

input : A GDS $\mathcal{S}(X, F, \mathcal{W}, K)$, VFA Scheme $vfaScheme$ to use.
output: Phase spaces, functional equivalence classes and cycle equivalence classes.

Instantiate: empty map $pidPhaseSpaceMap < PID, PhaseSpace >$, empty map $permMap < PID, perm >$, empty map $sysStMap < SSID, sysState >$, empty map $feecMap < feec, PIDSet >$, empty map $feecPhaseSpaceMap < feec, PhaseSpace >$;

Populate maps $permMap$ and $sysStMap$ (using schemes discussed in Section 7.1 and Algorithm 7) ;
if ($updateScheme == sequential$) **then**
    //Refer Algorithm 12.
    Compute VFA FEECs and store it in map $feecMap$;
    foreach *(entry in* $feecMap$*)* **do**
        //Each FEEC is represented using a PID.
        $vfaFEEC = entry.getFeec()$;
        Array $perm[1..n] = permMap.getPerm(vfaFEEC)$;
        //Compute the phase space for PID.
        foreach *(entry in* $sysStMap$*)* **do**
            $SSID = entry.getSSID()$;
            Array $systemState[1..n] = entry.getSystemState()$;
            Array $nextSystemState[1..n] =$
            $computeNextSysState(perm, systemState, \mathcal{W}, F, X)$;
            //Refer to Algorithm 16.
            $nextSSID = convertToSSID(nextSystemState)$;
        Insert $vfaFEEC$ and the above computed phase space (PS) into $pidPhaseSpaceMap$;
        //Refer to Algorithm 13.
        $computeAllFeecs(pidPhaseSpaceMap, feecMap, feecPhaseSpaceMap)$;
//Refer Algorithm 10.
$computeCEEC(feecMap, feecPhaseSpaceMap)$;
return ;

Algorithm 12: Serial algorithm to implement VFA FEEC computation. Algorithm 11 uses this algorithm. Refer Section 7.3 for a detailed description of this algorithm.

input : Dependency graph X, number of vertices n in X, VFA Scheme to be used.
output: VFA functional equivalence classes.

```
1  Instantiate: empty sets allPerms and derivedPerms ;
2  Initialize allPerms with PIDs of all possible permutation of vertices ;
3  if (vfaScheme == UpdateGraph) then
4      while (!allPerms.empty()) do
5          if (!derivedPerms.empty()) then
6              Erase all elements from set derivedPerms ;
7          //Returns a PID from the set allPerms and stores it in parentPID.
8          parentPID = allPerms.getAPID() ;
9          //Refer Algorithm 1.
10         derivedPerms = genVfaFeecsUpdateGraph(parentPID, n, X) ;
11         feecMap.insert(parentPID, derivedPerms) ;
12         Delete all PIDs in derivedPerms from allPerms set ;
13 else
14     if (vfaScheme == AcyclicOrientation) then
15         while (!allPerms.empty()) do
16             //Returns a PID from the set allPerms and stores it in PID.
17             PID = allPerms.getAPID() ;
18             //Refer Algorithm 3.
19             cannonPID = genCannonPID(PID, X, n) ;
20             //Update feecMap.
21             if (feecMap.containsKey(cannonPID)) then
22                 //feecMap already contains entry corresponding to cannonPID.
23                 //getPIDSetRef() returns reference to the existing set of functionally
                   equivalent PIDs.
24                 fePIDSet = feecMap.getPIDSetRef(cannonPID) ;
25                 fePIDSet.add(PID) ;
26             else
27                 //feecMap has no entry corresponding to PID. Hence add new class.
28                 Instantiate: fePIDSet ;
29                 fePIDSet.add(PID) ;
30                 feecMap.insert(PID, fePIDSet) ;
31             Delete PID from allPerms set ;
32 return ;
```

Algorithm 13: Serial algorithm used to identify if multiple VFA FEECs can be merged. Algorithm 11 uses this algorithm. Refer Section 7.3 for a detailed description of this algorithm.

input : *pidPhaseSpaceMap*, $feecMap$, *feecPhaseSpaceMap*. (Refer Section 7.1 for definition of these maps.)
output: FEEC that a VFA FEEC belongs to.

```
1  //This is a single PID that identifies a FEEC computed using VFA technique.
2  vfaFEEC = pidPhaseSpaceMap.getPID() ;
3  //Get Phase Space for vfaFEEC.
4  PS = pidPhaseSpaceMap.getPS(vfaFEEC) ;
5  Iterate over feecMap to get all keys from this map and store the keys in set feecSet
   //These keys are PIDs representing FEECs.
6  isInExistingClass = false ;
7  foreach (feec in feecSet) do
8      //Each element of feecSet is a PID representing a FEEC.
9      //Get Phase space corresponding to this FEEC.
10     feecPS = feecPhaseSpaceMap.getPS(feec) ;
11     if (PS == feecPS) then
12         //Equivalence class computed using VFA technique has the same phase space
           as that of another FEEC.
13         Add vfaFEEC and all PIDs belonging to this VFA FEEC to feecMap ;
14         feecMap.deleteEntry(vfaFEEC) ;
15         isInExistingClass = true ;
16         break ;
17 if (isInExistingClass == false) then
18     //Phase space of equivalence class computed using VFA technique does not match
       phase space of any other FEEC.
19     Insert < vfaFEEC, PS > into feecPhaseSpaceMap ;
20 //Clear all entries from the map.
21 pidPhaseSpaceMap.clearAll() ;
22 return ;
```

$O(|K|^n \cdot (n_e + n))$ time. Since $n_e = O(n^2)$ in the worst case, the time required to determine the phase space for all $n!$ PIDs is $O(|K|^n \cdot n^2 \cdot n!)$. Subroutine *computeFEEC*() is invoked for each PID to identify the FEEC that the PID belongs to. In each invocation of *computeFEEC*(), we iterate over all the existing keys in $feecMap$. In each iteration, the phase space of a $feec \in feecMap$ is compared to the phase space of PID. This comparison takes $O(|K|^n)$ time. Hence, an upper bound on the time taken by each invocation of *computeFEEC*() is $O(n_{FE} \cdot |K|^n)$. *computeFEEC*() is invoked $n!$ times. Thus, the total time taken to generate all the FEECs is the sum of the time taken to generate all $n!$ phase spaces and to determine the FEECs that these belong to. This time is $O(|K|^n \cdot n! \cdot (n^2 + n_{FE}))$. The time complexity for *computeCEEC*() (Algorithm 10) depends on the number of FEECs and on the time taken to compute the strongly connected components and the limit cycles of the phase space graph of each FEEC. Determination of the strongly connected components and the limit cycles for each $feec$ is linear in the size of $feecPS$, the phase space graph corresponding to $feec$. Since $feecPS$ contains $O(|K|^n)$ vertices and edges, the time taken to compute each of the n_{FE} limit cycles is $O(|K|^n)$. Once the limit cycle *csc* corresponding to a $feec$ is computed, we iterate over all existing entries in *ceecMap*, in the worst case, and compare *csc* to the cycle structure of each *ceec* $\in$ *ceecMap*. This allows us to determine the CEEC that a $feec$ belongs to. Comparison of two cycle structures take $O(n_{CS})$ time. Thus, an upper bound on the determination of the CEEC that a $feec$ belongs to, is $O(n_{CE} \cdot n_{CS})$. The time complexity of *computeFEEC*() is given by $O(n_{FE} \cdot (|K|^n + n_{CE} \cdot n_{CS}))$. The time complexity for the serial brute force GDS evaluation algorithm is the sum of the time required to compute all FEECs and all CEECs. This time is given by $O(|K|^n \cdot n! \cdot (n^2 + n_{FE}) + n_{FE} \cdot (|K|^n + n_{CE} \cdot n_{CS}))$. ■

Proposition 8 *The space complexity for the serial brute force GDS evaluation algorithm (Algorithm 8) is* $O(n \cdot n! + |K|^n \cdot (n + n_{FE}) + n_{CE} \cdot n_{CS})$.

Proof. The data structures *permMap*, *sysStMap*, *systemState*[], *nextSystemState*[] and *pidPhaseSpaceMap* in Algorithm 8 require $O(n \cdot n!)$, $O(|K|^n \cdot n)$, $O(n)$, $O(n)$ and $O(|K|^n)$ space respectively. Subroutine *computeFEEC*() requires $O(n!)$, $O(n_{FE})$, $O(|K|^n)$ and $O(n_{FE} \cdot |K|^n)$ space to store $feecMap$, $feecSet$, $feecPS$ and $feecPhaseSpaceMap$ respectively. An upper bound on the total space required by *computeFEEC*() over all $n!$ invocations is $O(n! + |K|^n \cdot n_{FE})$. Data structures *ceecMap* and $PIDCycleStructMap$ in subroutine *computeCEEC*() require $O(n_{CE} \cdot n_{cePIDs})$ and $O(n_{CE} \cdot n_{CS})$ space respectively. (The cycle structure of a phase space is stored in a vector of size $O(n_{CS})$.) Hence the total space complexity for serial brute force GDS evaluation is $O(n \cdot n! + |K|^n \cdot n + n_{FE} \cdot (n_{fePIDs} + |K|^n) + n_{CE} \cdot (n_{cePIDs} + n_{CS}))$. Note that $n_{FE} \cdot n_{fePIDs} = O(n!)$ and $n_{CE} \cdot n_{cePIDs} = O(n!)$. ■

B.1.3 Time and Space Complexity for Serial VFA GDS Evaluation Algorithm

Let n be the number of vertices in the dependency graph X, n_e be the number of edges in X, $n_{vfafePIDs}$ be the average number of PIDs in a single VFA FEEC, n_{vfaFE} be the number of FEECs computed using a VFA technique, n_{FE} be the number of FEECs, n_{CE} be the number of CEECs, n_{fePIDs} be the average number of PIDs in a single FEEC, n_{cePIDs} be the average number of PIDs in a single CEEC, n_{CS} be the length of the largest cycle among all phase space graphs of a GDS.

Proposition 9 *The time complexity of the serial VFA GDS evaluation algorithm (Algorithm 11) is* $O(n^3 \cdot n! + |K|^n \cdot n_{vfaFE} \cdot (n^2 + n_{vfaFE}) + n_{FE} \cdot (|K|^n + n_{CE} \cdot n_{CS}))$ *when update graph based VFA technique is used; and* $O(n^2 \cdot n! + |K|^n \cdot n_{vfaFE} \cdot (n^2 + n_{vfaFE}) + n_{FE} \cdot (|K|^n + n_{CE} \cdot n_{CS}))$ *when acyclic orientation-based technique is used.*

Proof. We assume that the time taken to add and delete elements to and from all map and set data structures used in Algorithms 11, 12, 13 and 10 is constant. Populating maps $permMap$ and $sysStMap$ take $O(n \cdot n!)$ and $O(n \cdot |K|^n)$ time respectively. The time taken to generate the VFA FEECs depends on the VFA technique used. If update graph-based VFA technique is used, then we iterate over n_{vfaFE} PIDs in the set $allPerms$ to generate the VFA FEECs. In each iteration, the subroutine $genVfaFeecsUpdateGraph()$ is invoked to generate a set of PIDs that belong to the same FEEC. This subroutine has a time complexity of $O(n^3 \cdot n_{vfafePIDs})$, as discussed in Section 3.1.2. Thus it takes $O(n^3 \cdot n_{vfafePIDs} \cdot n_{vfaFE})$ time to compute all VFA FEECs using update graph-based VFA technique. This time is $O(n^3 \cdot n!)$, in the worst case. Time taken to generate the VFA FEECs depends on the runtime of subroutine $genCannonPID()$, if acyclic orientation-based VFA technique is used. As discussed in Section 3.2.2, $genCannonPID()$ takes $O(n^2)$ time to generate the cannonical PID $cannonPID$ corresponding to a PID. Since $genCannonPID()$ is invoked $n!$ times, the time taken to generate the VFA FEECs using acyclic orientation-based approach is $O(n^2 \cdot n!)$. The VFA FEECs are stored in $feecMap$ and we iterate over all entries in $feecMap$ to generate the phase spaces of the VFA FEECs. The time taken to generate the phase space of a single VFA FEEC is equal to that taken to generate the phase space of a PID belonging to that VFA FEEC. This is because all PIDs belonging to the same VFA FEEC have the same phase space and each VFA FEEC is represented by a single PID belonging to it. As discussed in Section B.1.2, the time taken to generate the phase space of a single PID is $O(|K|^n \cdot (n_e + n))$, which is $O(|K|^n \cdot n^2)$ in the worst case. Since there are n_{vfaFE} VFA FEECs, we take $O(|K|^n \cdot n^2 \cdot n_{vfaFE})$ time to generate the phase spaces of all the VFA FEECs. The subroutine $computeAllFeecs()$ (Algorithm 13) is used to determine if multiple VFA FEECs can be merged or not. This subroutine is invoked n_{vfaFE} times. The time complexity for each invocation of this subroutine can be derived in a similar manner (refer to Section B.1.2) as that of the $computeFEEC()$ subroutine (Algorithm 9) used in serial brute force GDS evaluation algorithm (Algorithm 8). Since we iterate over $O(n_{vfaFE})$ entries

in *computeAllFeecs*(), the time complexity of this subroutine over all n_{vfaFE} invocations is $O(|K|^n \cdot {n_{vfaFE}}^2)$. The runtime for subroutine *computeCEEC*(), used in serial VFA GDS evaluation algorithm (Algorithm 11), is same as that of the same subroutine used in serial GDS evaluation algorithm (Algorithm 8). Hence the time complexity of the serial VFA GDS evaluation algorithm (Algorithm 11) is $O(n^3 \cdot n! + |K|^n \cdot n_{vfaFE} \cdot (n^2 + n_{vfaFE}) + n_{FE} \cdot (|K|^n + n_{CE} \cdot n_{CS}))$ when update graph based VFA technique is used; and $O(n^2 \cdot n! + |K|^n \cdot n_{vfaFE} \cdot (n^2 + n_{vfaFE}) + n_{FE} \cdot (|K|^n + n_{CE} \cdot n_{CS}))$ when acyclic orientation-based technique is used. Note that $n_{vfaFE} \geq n_{FE}$. ■

Proposition 10 *The space complexity for serial VFA GDS evaluation algorithm (Algorithm 11) is* $O(n! \cdot n + |K|^n \cdot (n_{vfaFE} + n) + n_{CE} \cdot n_{CS})$.

Proof. It takes $O(n!)$ space to generate all VFA FEECs (Algorithm 12). The data structures $permMap$, $sysStMap$, $systemState[]$, $nextSystemState[]$ and $pidPhaseSpaceMap$ in Algorithm 8 require $O(n \cdot n!)$, $O(|K|^n \cdot n)$, $O(n)$, $O(n)$ and $O(|K|^n)$ space respectively. Subroutine *computeAllFeecs*() requires $O(n!)$, $O(n_{vfaFE})$, $O(|K|^n)$ and $O(n_{vfaFE} \cdot |K|^n)$ space to store $feecMap$, $feecSet$, $feecPS$ and $feecPhaseSpaceMap$ respectively. An upper bound on the total space required by *computeAllFeecs*() over all n_{vfaFE} invocations is $O(n! + n_{vfaFE} \cdot |K|^n))$. Data structures $ceecMap$ and $PIDCycleStructMap$ in subroutine *computeCEEC*() require $O(n_{CE} \cdot n_{cePIDs})$ and $O(n_{CE} \cdot n_{CS})$ space respectively. (The cycle structure of a phase space is stored in a vector of size $O(n_{CS})$.) Hence the total space complexity for serial VFA GDS evaluation algorithm (Algorithm 11) is $O(n! \cdot n + |K|^n \cdot (n_{vfaFE} + n) + n_{CE} \cdot (n_{cePIDs} + n_{CS}))$. Note that $n_{CE} \cdot n_{cePIDs} = O(n!)$. ■

B.2 GDS Evaluation using Hadoop MapReduce

B.2.1 Memory Optimization Algorithms

B.2.2 Brute Force GDS Evaluation

B.2.2.1 Non-iterative Algorithm

B.2.2.2 Time and Space Complexity of Non-iterative Brute Force GDS Evaluation Algorithm

Let n be the number of vertices in the dependency graph X, n_e be the number of edges in the dependency graph X, $|K|$ be the number of vertex states of a single vertex, n_{cePIDs} be the average number of permutations that belong to a single cycle equivalence class, n_{fePIDs} be the average number of permutations that belong to a single functional equivalence class,

Algorithm 14: Algorithm to generate the actual permutation of vertices from its permutation index.

input : A permutation index denoted by PID and number of vertices n in the GDS.
output: Array $perm[1..n]$ containing the actual permutation of vertices corresponding to PID.

```
1  Instantiate: Array perm[1..n], Array fact[1..n] ;
2  k = 1, j = 0 ;
3  fact[k] = 1 ;
4  while (++k ≤ n) do
5      fact[k] = fact[k − 1] ∗ (k − 1) ;
6  for (k = 1; k ≤ n; ++k) do
7      perm[k] = pIdx/fact[n − k + 1] ;
8      pIdx = pIdx % fact[n − k + 1] ;
9  for (k = n; k > 1; −−k) do
10     for (j = k − 1; j ≥ 1; −−j) do
11         if (perm[j] ≤ perm[k]) then
12             perm[k]++ ;
13 return perm ;
```

Algorithm 15: Algorithm to generate a unique permutation index PID from a given permutation of vertices.

input : Array $perm[1..n]$, number of vertices n in the GDS.
output: A PID.

```
1  Instantiate: Array count[1..n] ;
2  for (i = 1; i ≤ n; i++) do
3      count[i] = 0 ;
4  numPerms = factorial(n) ;
5  PID = 0 ;
6  for (i = 1; i ≤ n; i++) do
7      count[perm[i]]++ ;
8  for (i = 1; i ≤ n; i++) do
9      count[i] += count[i − 1] ;
10 for (i = 1; i ≤ n; ++i) do
11     numPerms = numPerms/(n − i) ;
12     PID += (count[perm[i]] − 1) ∗ numPerms ;
13     for (i = perm[i]; i ≤ n; i++) do
14         count[i]−− ;
15 return PID ;
```

Algorithm 16: Algorithm to generate a unique SSID from a given GDS system state.

input : Array $systemState[1..n]$, number of vertices n in the dependency graph of the GDS and number of states ($radix$) that a vertex of a GDS can be in. $radix = |K|$, where K is the vertex state set.
output: A $SSID$.

```
1  SSID = 0, expo = 0 ;
2  for (itime = n; itime ≥ 1; −−itime) do
3      SSID += systemState[itime] ∗ radix^expo ;
4      ++expo ;
5  return SSID ;
```

Algorithm 17: Algorithm to generate a GDS system state from a unique SSID.

input : A $SSID$, number of vertices n in the dependency graph of the GDS and number of states ($radix$) that a vertex of a GDS can be in. $radix = |K|$, where K is the vertex state set.
output: Array $systemState[1..n]$.

```
1  Instantiate:Array systemState[1..n] ;
2  iElem = n ;
3  qt = currSysStIndex ;
4  if (SSID == 0) then
5  |  Set all elements of systemState to 0 ;
6  else
7  |  while (qt ≠ 0 and iElem ≥ 1) do
8  |  |  systemState[iElem] = qt%radix ;
9  |  |  qt = qt/radix ;
10 |  |  iElem − − ;
11 |  if (iElem > 1) then
12 |  |  while (iElem ≠ 1) do
13 |  |  |  systemState[iElem − −] = 0 ;
14 return systemState ;
```

Algorithm 18: Algorithm for non-iterative brute force GDS evaluation using MapReduce framework. Refer Section 8.2.1 for a detailed description of this algorithm.

input : A GDS $\mathcal{S}(X, F, \mathcal{W}, K)$.
output: Phase spaces, functional equivalence classes and cycle equivalence classes for the given GDS.

```
1  Read configuration file ;
2  Setup Hadoop configuration values for system state transition evaluation, FEEC
   computation and CEEC computation jobs ;
3  Generate input file SysStTranInpFile for system state transition evaluation job ;
4  //Submit system state transition evaluation job.
5  isJob_1_Succ = submitSystem_State_Transition_EvalJob() ;
6  if (isJob_1_Succ == true) then
7      //Submit FEEC computation job.
8      isJob_2_Succ = submitFEEC_ComputationJob() ;
9      if (isJob_2_Succ == true) then
10         //Submit CEEC computation job.
11         submitCEEC_ComputationJob() ;
12 return ;
```

n_{FE} be the number of FEECs, n_{CE} be the number of CEECs and n_{CS} be the length of the largest cycle among all phase space graphs of a GDS.

Let M be the number of map tasks and R be the number of reduce tasks being executed concurrently.

Proposition 11 *Time complexity for the map phase of the system state transition evaluation job is $O(n! \cdot |K|^n \cdot n^2/M)$. Time complexity for the reduce phase of the system state transition evaluation job is $O(n \cdot n! \cdot |K|^n \cdot log(|K|)/R)$.*

Proof. First we compute the time complexity for each *map*() function (Algorithm 19) of the system state transition evaluation job. Subroutine *getPerm*() takes $O(n^2)$ time to generate an actual permutation of vertices from a given PID. Subroutine *getSystemState*() takes $O(n)$ time to generate a GDS system state from a given $SSID$. *genSSID*() takes $O(n)$ time to generate a unique SSID from a given GDS system state. The function *computeNextSystemState*() takes $O(n_e)$ time to execute vertex functions on all vertices of a GDS and compute the next system state that the GDS transitions to. Since $n_e = O(n^2)$ in worst case, the time complexity for each *map*() function of the system state transition evaluation job is $O(n^2)$. This *map*() function is called $O(n! \cdot |K|^n)$ times. Since M map tasks execute concurrently, the time complexity for the map phase of the system state transition evaluation job is $O(n! \cdot |K|^n \cdot n^2/M)$.

Algorithm 19: Algorithm for map phase of system state transition evaluation MapReduce job. Refer Section 8.2.1.1 for a detailed description of this algorithm.

input : GDS $\mathcal{S}(X, F, \mathcal{W}, K)$ and a $< Key, Value >$ pair where Key is the position of a line in the input file of the MapReduce Job and $Value$ is the line of the input file.

output: An intermediate $(Key, Value)$ pair where Key is a PID and $Value$ is the corresponding system state transition $stateTransition$ of the GDS.

1 $//Value$ is a line of the input file.
2 // Each line of this Job's input file contains a (PID,SSID) pair.
3 $PID = Value.getPID()$;
4 $SSID = Value.getSSID()$;
5 //Generate the actual permutation of vertices and store it in array $perm$. Refer to algorithm 14.
6 $Array\ perm[1..n] = getPerm(PID)$;
7 //Refer algorithm 17.
8 $Array\ sysState[1..n] = getSystemState(SSID)$;
9 $Array\ nextSysState[1..n] = computeNextSystemState(perm, sysState, \mathcal{W}, F, X)$;
10 //Refer algorithm 16.
11 $nextSSID = genSSID(nextSysState)$;
12 $stateTransition = SSID.toString() + " " + nextSSID.toString()$;
13 $emit(PID, stateTransition)$;
14 **return** ;

Algorithm 20: Algorithm for reduce phase of system state transition evaluation MapReduce job. Refer Section 8.2.1.1 for a detailed description of this algorithm.

input : A $<Key, Value>$ pair with a permutation index PID as the Key and a list $SysStateTransList$ as the $Value$. $SysStateTransList$ consists of all possible system state transitions for the key PID.

output: A $<Key, Value>$ pair consisting of a PID and a string representing the phase space of the PID.

1 **Instantiate:** empty map $stateTransMap$, which is kept sorted by its keys, and empty string $sortedStateTrans$;

2 **foreach** *(sysStTrans in SysStateTransList)* **do**

3 $fromSSID = sysStTran.getFromState()$;

4 $toSSID = sysStTran.getToState()$;

5 $stateTransMap.insert(fromSSID, toSSID)$;

6 //Iterate over all records in sorted map $stateTransMap$.

7 **foreach** *((Key,Value) pair* $< fromSSID, toSSID >$ *in stateTransMap)* **do**

8 $sortedStateTrans = sortedStateTrans +$" "$+toSSID.toString()$;

9 $emit(sortedStateTrans, PID)$;

10 **return** ;

Algorithm 21: Algorithm for map phase of FEEC computation job. Refer Section 8.2.1.2 for a detailed description of this algorithm.

input : A $<Key, Value>$ pair where Key is the position of a line in the input file of the MapReduce Job and $Value$ is the line of the input file.

output: An intermediate $<Key, Value>$ pair where Key is a string representing the phase space of a PID and $Value$ is a PID.

1 //$Value$ is a line of the input file.

2 // Each line of the input file contains a PID and a string representation of the phase space for the PID.

3 $phaseSpace = Value.getPhaseSpace()$;

4 $PID = Value.getPID()$;

5 $emit(phaseSpace, PID)$;

6 **return** ;

Algorithm 22: Algorithm for reduce phase of FEEC computation MapReduce job. Refer Section 8.2.1.2 for a detailed description of this algorithm.

input : A $<Key, Value>$ pair where Key is a string $phaseSpaceString$ representing a phase space of the given GDS and $Value$ is a list $PIDList$ containing all PIDs having the same $phaseSpaceString$.

output: A $<Key, Value>$ pair, where the $phaseSpaceString$ is the Key and the $Value$ is a string obtained by concatenating all PIDs having the same $phaseSpaceString$.

```
1 Instantiate: empty string feecPerms ;
2 foreach (pid in PIDList ) do
3     feecPerms = feecPerms + " " + pid.toString() ;
4 emit(phaseSpaceString, feecPerms) ;
5 return ;
```

Algorithm 23: Algorithm for map phase of CEEC computation job. Section 8.2.1.3 discusses this algorithm in detail.

input : A $<Key, Value>$ pair where the position of a line in the input file of the MapReduce Job forms the Key and the line of the input file forms the $Value$.

output: An intermediate $<Key, Value>$ pair, where Key is the cycle structure for a functional equivalence class and $Value$ is a string of functionally equivalent PIDs.

```
1  Instantiate: empty string feecPerms, empty string cycleStruct ;
2  //Value is a line of the input file.
3  //Each line of the input file contains two fields. The first is a string representing the
   phase space of a set of functionally equivalent PIDs and the second is the set of
   functionally equivalent PIDs.
4  phaseSpace = Value.getPhaseSpace() ;
5  feecPerms = Value.getFEECPerms() ;
6  //Fetch a single permutation from the string of functionally equivalent permutations.
7  feecPid =feecPerms.getAPermutation() ;
8  Compute cycle structure for permutation feecPid and store it in cycleStruct ;
9  emit(cycleStruct, feecPerms) ;
10 return ;
```

Algorithm 24: Algorithm for reduce phase of CEEC computation job. Section 8.2.1.3 discusses this algorithm in detail.

input : A $<Key, Value>$ pair . The cycle structure *cycleStruct* of a functional equivalence class forms the Key. The $Value$ is a list $CEECList$ of all functional equivalence classes that have the same *cycleStruct*.

output: A $<Key, Value>$ pair. The cycle structure *cycleStruct* forms the Key. The $Value$ is a string obtained by concatenating all PIDs having isomorphic cycle structures.

```
1 Instantiate: empty string ceecPerms ;
2 foreach (element ∈ CEECList) do
3 |  ceecPerms = ceecPerms +" "+ element ;
4 emit(cycleStruct,ceecPerms) ;
5 return ;
```

Now we compute the time complexity for each *reduce*() function (Algorithm 20) of the system state transition evaluation job. In each *reduce*() function, we iterate over all system state transitions in $SysStateTransList$ and extract $fromSSID$ and $toSSID$ values from each of these state transitions. $SysStateTransList$ contains $O(|K|^n)$ system state transitions in it. Then we insert these $<fromSSID, toSSID>$ values into $stateTransMap$. Since $stateTransMap$ contains $O(|K|^n)$ key-value pairs and is kept sorted by its keys, insertion of key-value pairs to this map takes $O(log(|K|^n))$ time. Generation of the string $sortedStateTrans$ requires iterating over all key-value pairs in $stateTransMap$ and takes $O(|K|^n)$ time. Hence, the time complexity for each *reduce*() function is $O(|K|^n \cdot log(|K|^n))$. This function is called $n!$ times. Therefore, the time complexity for the reduce phase of the system state transition evaluation job is $O(n! \cdot |K|^n \cdot log(|K|^n)/R)$. ■

Proposition 12 *Space complexity for the map phase of the system state transition evaluation job is* $O(n^2 \cdot M + n! \cdot |K|^n)$. *Space complexity for the reduce phase of the system state transition evaluation job is* $O(|K|^n \cdot R + n! \cdot |K|^n)$.

Proof. Arrays *perm*[], *sysState*[] and *nextSysState*[] in each *map*() function (Algorithm 19) of the system state transition evaluation job require $O(n)$ storage space. Subroutines *getPerm*(), *getSystemState*() and *genSSID*() have $O(n)$ space complexity. The dependency graph X is stored as an adjacency list and requires $O(n+n_e)$ storage. Since $n_e = O(n^2)$ in the worst case, the space complexity for each *map*() function is $O(n^2)$. A *map*() function is called $O(n! \cdot |K|^n)$ times. This produces $O(n! \cdot |K|^n)$ key-value pairs, each of which requires constant space. Hence the space complexity for the map phase of the system state transition evaluation job is $O(n^2 \cdot M + n! \cdot |K|^n)$.

Each of the data structures $stateTransMap$, $sortedStateTrans$ and $SysStateTransList$ in

a *reduce*() function (Algorithm 20) of the system state transition evaluation job requires $O(|K|^n)$ storage space. Hence the space complexity for each *reduce*() function is $O(|K|^n))$. The reduce phase produces $n!$ key-value pairs, each of which requires $O(|K|^n)$ storage space. With R concurrent reduce tasks, the space complexity for the reduce phase is $O(|K|^n \cdot R + n! \cdot |K|^n)$. ■

Proposition 13 *Time complexity for the map phase of the FEEC computation job is* $O(n!/M)$. *Time complexity for the reduce phase of the FEEC computation job is* $O(n!/R)$.

Proof. Each *map*() function (Algorithm 21) of FEEC computation job takes constant time to extract *phaseSpace* and PID from $Value$ and output *phaseSpace* and PID. This function is called $n!$ times in total. Since M map tasks are being executed in parallel, the time complexity for the map phase of FEEC computation job is $O(n!/M)$.

The list $PIDList$ used in each *reduce*() function (Algorithm 22) of FEEC computation job contains n_{fePIDs} PIDs in it. We iterate over $PIDList$ to generate $feecPerms$. Assuming that fetching a *pid* from $PIDList$ takes constant time, the time complexity for each *reduce*() function is $O(n_{fePIDs})$. This *reduce*() function is called n_{FE} number of times. Since R reduce tasks execute concurrently, the time complexity for the reduce phase of FEEC computation job is $O(n_{fePIDs} \cdot n_{FE}/R)$, which is $O(n!/R)$. ■

Proposition 14 *Space complexity for the map phase of the FEEC computation job is* $O(|K|^n \cdot (M + n!))$. *Space complexity for the reduce phase of the FEEC computation job is* $O(R \cdot (n_{fePIDs} + |K|^n) + n_{FE} \cdot |K|^n + n!)$.

Proof. The space complexity for each *map*() function (Algorithm 21) of FEEC computation job depends on the space required to store *phaseSpace* string. Since *phaseSpace* requires $O(|K|^n)$ space, space complexity for each *map*() function is $O(|K|^n)$. This *map*() function is called $n!$ times, which produces $n!$ key-value pairs. Each key-value pair requires $O(|K|^n)$ space. With M map tasks being executed concurrently, the total space complexity for the map phase of the FEEC computation job is $O(|K|^n \cdot (M + n!))$.

Each *reduce*() function (Algorithm 22) of FEEC computation job requires $O(n_{fePIDs})$ space to store $PIDList$ and $O(|K|^n)$ space to store *phaseSpaceString*. The total space complexity for each *reduce*() function is $O(n_{fePIDs} + |K|^n)$. The reduce phase of this job produces n_{FE} number of key-value pairs, each of which requires $O(|K|^n + n_{fePIDs})$ space. With R reduce tasks being executed concurrently, the space complexity for the reduce phase is $O(R \cdot (n_{fePIDs} + |K|^n) + n_{FE} \cdot (|K|^n + n_{fePIDs}))$, which is $O(R \cdot (n_{fePIDs} + |K|^n) + n_{FE} \cdot |K|^n + n!)$. ■

Proposition 15 *Time complexity for the map phase of the CEEC computation job is* $O(|K|^n \cdot n_{FE}/M)$. *Time complexity for the reduce phase of the CEEC computation job is* $O(n!/R)$.

Proof. The time complexity for each *map*() function (Algorithm 23) of the CEEC computation job depends on the time taken to compute the cycle structure *cycleStruct* of the phase space graph of PID $feecPid$. Computing this cycle structure involves determining the number of strongly connected components and the cycle lengths of these components in the phase space graph of $feecPid$. This phase space graph contains $O(|K|^n)$ vertices and $O(|K|^n)$ edges and its cycle structure can be obtained in $O(|K|^n)$ time. Therefore, the time complexity for each *map*() function is $O(|K|^n)$. Since this *map*() function is called n_{FE} times, the time complexity for the map phase of the CEEC computation job is $O(|K|^n \cdot n_{FE}/M)$.

Each *reduce*() function (Algorithm 24) iterates over all PIDs belonging to the same CEEC to generate the string $ceecPerms$. Thus the time complexity for each *reduce*() function is $O(n_{cePIDs})$. This function is called n_{CE} times. Hence the time complexity for the reduce phase of CEEC computation job is $O(n_{cePIDs} \cdot n_{CE}/R)$, which is $O(n!/R)$. ■

Proposition 16 *Space complexity for the map phase of the CEEC computation job is* $O((|K|^n + n_{fePIDs}) \cdot M + n_{FE} \cdot n_{CS} + n!)$. *Space complexity for the reduce phase of the CEEC computation job is* $O(n_{cePIDs} \cdot R + n_{CE} \cdot n_{CS} + n!)$.

Proof. Each *map*() function of the CEEC computation job requires $O(|K|^n)$ space to store *phaseSpace* and $O(n_{fePIDs})$ to store $feecPerms$. Computing the cycle structure of PID $feecPid$ requires $O(|K|^n)$ storage to store the data structures for the phase space graph of $feecPid$, the number of strongly connected components n_{CC} in it and the set of cycle lengths in it. Hence, the space complexity for each *map*() function is $O(|K|^n + n_{fePIDs})$. This *map*() function is called n_{FE} number of times to produce n_{FE} key-value pairs. Each key-value pair $<cycleStruct, feecPerms>$ requires $O(n_{CS} + n_{fePIDs})$ space, since *cycleStruct* is stored in a vector of size $O(n_{CS})$. Thus the space complexity for the map phase of CEEC computation job is $O((|K|^n + n_{fePIDs}) \cdot M + n_{FE} \cdot (n_{CS} + n_{fePIDs}))$, which is $O((|K|^n + n_{fePIDs}) \cdot M + n_{FE} \cdot n_{CS} + n!)$.

The space complexity for each *reduce*() function of CEEC computation job is $O(n_{cePIDs})$. The reduce phase produces n_{CE} key-value pairs, each of which requires $O(n_{CS} + n_{cePIDs})$ space. With R concurrent reduce tasks, the space complexity of the reduce phase is $O(n_{cePIDs} \cdot R + n_{CE} \cdot (n_{CS} + n_{cePIDs}))$, which is $O(n_{cePIDs} \cdot R + n_{CE} \cdot n_{CS} + n!)$. ■

B.2.2.3 System-state based Iterative Algorithm

B.2.2.4 Time and Space Complexity of System-state based Iterative Algorithm

Let n be the number of vertices in the dependency graph X, n_e be the number of edges in the dependency graph X, $|K|$ be the number of vertex states of a single vertex, n_{FE} be the number of FEECs and n_{fePIDs} be the average number of PIDs belonging to a single FEEC of a given GDS. Let M be the number of map tasks and R be the number of reduce tasks being executed concurrently.

Algorithm 25: System state-based iterative algorithm for brute force GDS evaluation. Refer Sections 8.2.2 and 8.2.2.1 for a detailed description of this algorithm.

input : A GDS $\mathcal{S}(X, F, \mathcal{W}, K)$.
output: Phase spaces, functional equivalence classes and cycle equivalence classes for the given GDS.

```
1  Read configuration file ;
2  Setup Hadoop configuration values for iterative FEEC computation MapReduce jobs,
   phase space computation job and CEEC computation job ;
3  Generate input file IterMrInpFile for first iteration of iterative FEEC computation
   MapReduce jobs ;
4  //Algorithm 26 used to implement method submitIterJobs().
5  isJob_1_Succ = submitIterJobs(IterMrInpFile) ;
6  if (isJob_1_Succ == true) then
7  |  //Submit Phase space computation job.
8  |  isJob_2_Succ = submitPhaseSpace_ComputationJob() ;
9  |  if (isJob_2_Succ == true) then
10 |  |  //Submit CEEC Computation Job.
11 |  |  submitCEEC_ComputationJob() ;
12 return ;
```

Proposition 17 *The time complexity for the map phase of the phase space computation job is $O(K^n \cdot n^2 \cdot n_{FE}/M)$. The time complexity for the reduce phase of the phase space computation job is $O(n_{FE}/M)$.*

Proof. First we determine the time complexity for each *map*() function (Algorithm 27) of the phase space computation job. Subroutines *getPerm*(), *getSystemState*(), *computeNextSystemState*() and *genSSID*() take $O(n^2)$, $O(n)$, $O(n_e)$ and $O(n)$ time respectively. The total number of system states *numAllSystemStates* for a given GDS is $O(|K|^n)$. Since $n_e = O(n^2)$ in the worst case, time complexity for each *map*() function is $O(K^n \cdot n^2)$. This *map*() function is called n_{FE} times. Hence the time complexity for the *map* phase of the phase space computation job is $O(K^n \cdot n^2 \cdot n_{FE}/M)$.

Now we determine the time complexity for each *reduce*() function of the phase space computation job. Each *reduce*() function takes constant time to execute. Since this function is called n_{FE} times, the reduce phase of the phase space computation job takes $O(n_{FE}/M)$ time. ■

Proposition 18 *The space complexity for the map phase of the phase space computation job is $O((n_{fePIDs} + |K|^n + n^2) \cdot M + n_{FE} \cdot |K|^n + n!)$. The space complexity for the reduce phase of the phase space computation job is $O((|K|^n + n_{fePIDs}) \cdot R + n_{FE} \cdot |K|^n + n!)$.*

Algorithm 26: Algorithm used to submit iterative FEEC computation jobs. Algorithm 25 uses this algorithm. Refer Sections 8.2.2 and 8.2.2.1 for a detailed description of this algorithm.

input : Input file $IterMrInpFile$ for first iteration of iterative FEEC computation MapReduce jobs.
output: All functional equivalence classes.

```
1  Instantiate: Set SSID = 0 and numTotSysStates as the total number of GDS
   system states ;
2  //We submit MapReduce jobs iteratively.
3  while (SSID < numTotSysStates ) do
4      Set Hadoop configuration parameters for current job ;
5      Set nextSSID = SSID +1 ;
6      //Submit iterative FEEC computation job.
7      isSucc = submitIter_FEEC_Comp_Job(SSID, nextSSID) ;
8      if (isSucc == true) then
9          //ifMultiPermKeys() returns true when reduce tasks run in the previous
           iteration outputs at least one key which consists of multiple permutations.
           Else it returns false.
10         if (ifMultiPermKeys() == true) then
11             SSID++ ;
12         else
13             break ;
14 Merge "relevant" output from iterative MapReduce jobs and feed them as an input to
   the Phase Space Computation job ;
15 return ;
```

Algorithm 27: Algorithm for map phase of phase space computation job. Section 8.2.2.3 describes this algorithm in detail.

input : GDS $\mathcal{S}(X, F, \mathcal{W}, K)$ and a $<Key, Value>$ pair, where a line of the input file forms the $Value$ and its offset forms the Key.

output: A set of functionally equivalent PIDs and their phase space as a key-value pair.

```
1  Instantiate: empty string sysStateTransStr ;
2  Set numAllSystemStates as number of possible GDS system states ;
3  //Value is a line of the input file.
4  //Each line consists of a string of functionally equivalent PIDs.
5  pidString = Value ;
6  //Extract a PID from pidString.
7  PID = pidString.getAPID() ;
8  //Generate actual permutation of vertices and store it in array perm.
9  Array perm[1..n] = getPerm(PID) ;
10 for (ssid = 0; ssid < numAllSystemStates; ssid++) do
11     //Generates the state of each vertex of the GDS and stores it in array sysState.
12     Array sysState[1..n] = getSystemState(ssid) ;
13     //Compute next system state that GDS transitions to.
14     Array toSysState[1..n] = computeNextSystemState(perm, sysState, W, F, X) ;
15     //Generate the system state index toSSID that uniquely identifies the system
       state array toSysState.
16     toSysState = genSSID(toSysState) ;
17     sysStateTransStr = sysStateTransStr+" "+(toSysState) ;
18 emit(stTranStr, pidString) ;
19 return ;
```

Proof. First we determine the space complexity for each each *map*() function (Algorithm 27) of the phase space computation job. Arrays *perm*[], *sysState*[] and *toSysState*[] take $O(n)$ space. *pidString* is a string of functionally equivalent PIDs and requires $O(n_{fePIDs})$ space. *sysStateTransStr* contains all system state transitions for a FEEC and requires $O(|K|^n)$ space. Space complexity for subroutines *getPerm*(), *genSSID*() and *getSystemState*() is $O(n)$. The dependency graph X is stored as an adjacency list and has a space complexity of $O(n + n_e)$. $n_e = O(n^2)$, in the worst case. Ignoring all lower order terms, the space complexity for each *map*() function is $O(n_{fePIDs} + |K|^n + n^2)$. The map phase produces n_{FE} key-value pairs, each of which requires $O(|K|^n + n_{fePIDs})$ space. Since M map tasks execute concurrently, the space complexity for the map phase of the phase space computation job is $O((n_{fePIDs} + |K|^n + n^2) \cdot M + n_{FE} \cdot (|K|^n + n_{fePIDs}))$, which is $O((n_{fePIDs} + |K|^n + n^2) \cdot M + n_{FE} \cdot |K|^n + n!)$.

Since each *reduce*() function of the phase space computation job writes the phase space of a FEEC and the equivalence class itself, it requires $O(|K|^n + n_{fePIDs})$ space to store these values. Each *reduce*() function is called n_{FE} times thereby producing n_{FE} key-value pairs which require a total of $O(n_{FE} \cdot (|K|^n + n_{fePIDs}))$ space. With R reduce tasks being executed concurrently, the space complexity for the reduce phase of the phase space computation job is $O((|K|^n + n_{fePIDs}) \cdot R + n_{FE} \cdot (|K|^n + n_{fePIDs}))$, which is $O((|K|^n + n_{fePIDs}) \cdot R + n_{FE} \cdot |K|^n + n!)$. ■

B.2.3 VFA GDS Evaluation

B.2.3.1 Non-iterative Algorithm

B.2.3.2 Time and Space Complexity of Non-iterative VFA GDS Evaluation Algorithm

Let n be the number of vertices in the dependency graph X, n_e be the number of edges in the dependency graph X, $|K|$ be the number of vertex states of a single vertex of a given GDS. Let $n_{vfafePIDs}$ be the average number of permutations that belong to a single VFA FEEC (i.e. a FEEC computed using a VFA technique) and n_{vfaFE} be the number of FEECs computed using a suitable VFA technique in a GDS.

Let M be the number of map tasks and R be the number of reduce tasks being executed concurrently.

Proposition 19 *The time complexity for the map phase of the VFA FEEC computation job, when it uses either VFA scheme 1 or VFA scheme 2, is given by* $O(n^3 \cdot n_{vfafePIDs} \cdot n!/M)$. *The time complexity for the map phase of the VFA FEEC computation job, when it uses VFA scheme 3, is given by* $O(n^2 \cdot n!/M)$.

Proof. First we determine the time complexity for each *map*() function (Algorithm 29) of

Algorithm 28: Algorithm for non-iterative VFA GDS evaluation using MapReduce framework. Refer Section 8.3.1 for a detailed description of this algorithm.

input : A GDS $\mathcal{S}(X, F, \mathcal{W}, K)$, VFA Scheme Id $vfaScheme$.
output: All phase spaces, functional equivalence classes and cycle equivalence classes for the given GDS.

```
1  Read configuration file ;
2  Setup Hadoop configuration values for VFA FEEC computation, system state
   transition evaluation, phase space based FEEC computation and CEEC computation
   jobs ;
3  Generate input file VFA_FEECJobInpFile for VFA FEEC computation job ;
4  //Submit VFA FEEC computation job.
5  if (vfaScheme == 0) then
6  |  //Submit VFA FEEC computation (with all vertex flips) job.
7  |  isJobSucc = submitVFAFEEC_AllVertexFlips() ;
8  else
9  |  if (vfaScheme == 1) then
10 |  |  //Submit VFA FEEC computation (with selective vertex flips) job.
11 |  |  isJobSucc = submitVFAFEEC_SelVertexFlips() ;
12 |  else
13 |  |  //Submit VFA FEEC computation (with acyclic orientations) job.
14 |  |  isJobSucc = submitVFAFEEC_Acyc() ;
15 if (isJobSucc == true ) then
16 |  //Submit system state transition evaluation job.
17 |  isJobSucc =submitSystem_State_Transition_EvalJob() ;
18 |  if (isJobSucc == true) then
19 |  |  //Submit phase space based FEEC computation job.
20 |  |  isJobSucc = submitPS_FEEC_ComputationJob() ;
21 |  |  if (isJobSucc == true) then
22 |  |  |  //Submit CEEC computation job.
23 |  |  |  submitCEEC_ComputationJob() ;
24 return ;
```

Algorithm 29: Algorithm for map phase of VFA FEEC computation (using update graph) job. This algorithm processes all possible PIDs of the given GDS to generate their FEECs. VFA scheme 1, discussed in Section 8.3.1.1, uses this algorithm for implementing the *map*() function.

input : The number of vertices n in the dependency graph X and a $<Key, Value>$ pair, where the offset of a line of the input file forms the Key and the line itself forms the $Value$.

output: An intermediate $<Key, Value>$ pair with a PID as the key and a set of functionally equivalent PIDs as the value.

```
1  //The set sortedDerPerms is a set of permutation indices (PIDs) that are
   functionally equivalent to each other. The elements in this set are ordered by their
   PIDs.
2  Instantiate: empty set sortedDerPerms ;
3  //Value is a line of the input file.
4  //Each line of the input file of this job has a unique PID.
5  PID = Value.getPID() ;
6  //See Algorithm 1.
7  sortedDerPerms = genFEPerms_UpdateGraph(PID, n) ;
8  Iterate over the elements of sortedDerPerms and concatenate them together to form
   a string vfaFEECPerms of functionally equivalent permutation indices ;
9  //vfaFEECPerms has all permutation indices in the sorted order.
10 emit(vfaFEECPerms, PID) ;
11 return ;
```

Algorithm 30: Algorithm for reduce phase of VFA FEEC computation (using update graph) job. VFA schemes 1 and 2, discussed in Section 8.3.1.1, use this algorithm for implementing their *reduce*() function.

input : A $<Key, Value>$ pair. The Key is a string $vfaFEECPerms$ obtained by concatenating functionally equivalent PIDs. The $Value$ is a PID.

output: A $<Key, Value>$ pair, where the Key is the same as the input Key and the $Value$ is a system state index $SSID$.

```
1  Set numTotalSysStates as the total number of possible system states for the given
   GDS ;
2  for (i = 0; i < numTotalSysStates; i++) do
3      emit(vfaFEECPerms, i) ;
4  return ;
```

Algorithm 31: Algorithm for map phase of VFA FEEC computation (using update graph) job. This algorithm processes selective PIDs of the given GDS to generate their FEECs. VFA scheme 2, discussed in Section 8.3.1.1, uses this algorithm for implementing the *map*() function.

input : The number of vertices n in the dependency graph X and a $<Key, Value>$ pair, where a line of the input file forms the $Value$ and its offset forms the Key.(Each line of the input file of this job has a unique PID).

output: An intermediate <Key,Value> pair with a PID as the key and a set of functionally equivalent PIDs as the value.

```
1  //The set sortedDerPerms is a set of permutation indices (PIDs) that are
   functionally equivalent to each other. The elements in this set are ordered by their
   PIDs.
2  Instantiate: empty set sortedDerPerms ;
3  PID = Value.getPID() ;
4  //See Algorithm 32.
5  sortedDerPerms = genFEPerms_UpdateGraph(PID, n) ;
6  if (sortedDerPerms! = null) then
7  |  Iterate over the elements of sortedDerPerms and concatenate them together to
   |  form a string vfaFEECPerms of functionally equivalent permutation indices ;
8  |  //vfaFEECPerms has all permutation indices in the sorted order.
9  |  emit(vfaFEECPerms, PID) ;
10 return ;
```

Algorithm 32: Generation of all functionally equivalent permutations from a given permutation, using vertex flips. This algorithm stops processing a given permutation if it generates a functionally equivalent permutation having an index lesser than that of the given permutation. Algorithm 31 and Section 8.3.1.1 describe the use of this algorithm.

input : A permutation index $parent\_PIdx$ that uniquely identifies a permutation of vertices and size n of the GDS dependency graph.

output: A set of permutation indices which are functionally equivalent to the input permutation index $parent\_PIdx$.

```
1  //Set sortedDerPerms contains all permutations that are functinally equivalent to
   the parent_PIdx. These permutations are sorted by their permutation indices.
2  // Queue q_DerPerms stores functionally equivalent permutations in the order in
   which they are derived from the input permutation parent_PIdx.
3  //Arrays parentPerm stores actual permutation of vertices identified by a unique
   permutation index.
4  Instantiate: empty queue q_DerPerms, empty set sortedDerPerms, permutation
   index swappedPermIndex, array parentPerm[1..n] ;
5  sortedDerPerms.add(parent_PIdx) ;
6  q_DerPerms.enqueue(parent_PIdx) ;
7  while (!q_DerPerms.empty()) do
8      candidatePermIndex = q_DerPerms.dequeue() ;
9      //getPerm() generates the actual permutation of vertices given a permutation
       index candidatePermIndex.
10     parentPermi[1..n] = getPerm(candidatePermIndex) ;
11     for (j = 1;j ≤ (n − 1);j + +) do
12         if (π[i],π[i + 1]) ∉ e[X] then
13             //See Algorithm 2.
14             swappedPermIndex = genSwappedPerm(parentPerm, i) ;
15             if (swappedPermIndex < candidatePermIndex ) then
16                 return null ;
17             else
18                 if (!sortedDerPerms.contains(swappedPermIndex)) then
19                     sortedDerPerms.add(swappedPermIndex) ;
20                     q_DerPerms.enqueue(swappedPermIndex) ;
21 return sortedDerPerms ;
```

Algorithm 33: Algorithm for map phase of VFA FEEC computation (using acyclic orientations) job. VFA scheme 3, discussed in Section 8.3.1.1, uses this algorithm for implementing the *map*() function.

input : A $<Key, Value>$ pair, where the offset of a line of the input file forms the Key and the line itself forms the $Value$.
output: A $<Key, Value>$ pair with a PID as the $Value$ and its cannonical permutation $cannonPID$ as the Key.

```
1 //Value is a line of the input file.
2 //Each line of the input file of this job has a unique PID.
3 PID = Value.getPID() ;
4 //See Algorithm 3.
5 cannonPID = genCannonPerm(PID) ;
6 emit(cannonPID, PID) ;
7 return ;
```

Algorithm 34: Algorithm for reduce phase of VFA FEEC computation (using acyclic orientations) job. VFA scheme 3, discussed in Section 8.3.1.1, uses this algorithm for implementing the *reduce*() function.

input : A $<Key, Value>$ pair where Key is a cannonical permutation and $Value$ is a list $PIDList$ of permutation indices having the same cannonical permutation.
output: A $<Key, Value>$ pair, where the Key is a string containing functionally equivalent PIDs and $Value$ is a system state index $SSID$.

```
1 Instantiate: empty string vfaFEECPerms ;
2 Set numTotalSysStates as the total number of possible system states for the given
  GDS ;
3 foreach (PID in PIDList) do
4 |  feecPerms = feecPerms +" "+ PID ;
5 for (i = 0; i < numTotalSysStates; i++) do
6 |  emit(feecPerms, i) ;
7 return ;
```

Algorithm 35: Algorithm for map phase of system state transition evaluation MapReduce job. Refer Section 8.3.1.2 for a detailed description of this algorithm.

input : GDS $\mathcal{S}(X, F, \mathcal{W}, K)$ and a $<Key, Value>$ pair where Key is the position of a line in the input file of the MapReduce Job and $Value$ is the line of the input file. Each line of the input file contains two entries. The first entry is a string $vfaFEECPerms$ containing functionally equivalent PIDs, and is an output from the VFA FEEC computation job. The second entry is a SSID.

output: An intermediate $(Key, Value)$ pair where Key is the input string $vfaFEECPerms$ and $Value$ is a system state transition of the GDS.

1 //Get the string of functionally equivalent PIDs.
2 $vfaFEECPerms = Value.getPIDs()$;
3 //Get a PID from the string $vfaFEECPerms$.
4 $PID = vfaFEECPerms.getAPID()$;
5 $SSID = Value.getSSID()$;
6 //Generate the actual permutation of vertices and store it in array *perm*.
7 $Array\ perm[1..n] = getPerm(PID)$;
8 //Generate the state of each vertex of the GDS and store it in the array *sysState*.
9 $Array\ sysState[1..n] = getSystemState(SSID)$;
10 $Array\ nextSysState[1..n] = computeNextSystemState(perm, sysState, \mathcal{W}, F, X)$;
11 //Generate the system state index $nextSSID$ that uniquely identifies the system state array *nextSysState*.
12 $nextSSID = genSSID(nextSysState)$;
13 $stateTransition = SSID.toString() + " " + nextSSID.toString()$;
14 $emit(vfaFEECPerms, stateTransition)$;
15 **return** ;

Algorithm 36: Algorithm for reduce phase of system state transition evaluation MapReduce job. Refer Section 8.3.1.2 for a detailed description of this algorithm.

input : A $<Key, Value>$ pair with a string of functionally equivalent permutation indices $fePIDs$ as the Key and a list $SysStateTransList$ as the $Value$. $SysStateTransList$ consists of all possible system state transitions for the key $fePIDs$.

output: A $<Key, Value>$ pair. The Key is a string of functionally equivalent PIDs. The $Value$ is a string representing the phase space of these functionally equivalent PIDs.

1 **Instantiate:** empty map $stateTransMap$, which is kept sorted by its keys ;
2 **Instantiate:** empty string $sortedStateTrans$;
3 **foreach** *(sysStTrans in SysStateTransList)* **do**
4 $\quad fromSSID = sysStTran.getFromState()$;
5 $\quad toSSID = sysStTran.getToState()$;
6 $\quad stateTransMap.insert(fromSSID, toSSID)$;
7 //Iterate over all records in sorted map $stateTransMap$.
8 **foreach** *((Key,Value) pair* $< fromSSID, toSSID >$ *in stateTransMap)* **do**
9 $\quad sortedStateTrans = sortedStateTrans +$" "$+toSSID.toString()$;
10 $emit(sortedStateTrans, fePIDs)$;
11 **return** ;

the VFA FEEC computation job, when it uses VFA scheme 1. (Refer to Section 8.3.1.1). As discussed in Section 3.1.2, subroutine $genFEPerms\_UpdateGraph()$ in each $map()$ function takes $O(n^3 \cdot n_{vfafePIDs})$ time to generate the set $sortedDerPerms$ of permutations that are functionally equivalent to PID. Since $sortedDerPerms$ has $O(n_{vfafePIDs})$ in it, generation of string $vfaFEECPerms$ takes $O(n_{vfafePIDs})$ time. This $map()$ function is called $O(n!)$ times. With M map tasks being executed concurrently, the time complexity for the map phase of the VFA FEEC computation job, using VFA scheme 1, is $O(n^3 \cdot n_{vfafePIDs} \cdot n!/M)$. This is also the worst case time complexity of the VFA FEEC computation job when it uses VFA scheme 2.

Now we determine the time complexity for each $map()$ function (Algorithm 33) of the VFA FEEC computation job, when it uses VFA scheme 3. This time complexity depends on the time complexity of the subroutine $genCannonPerm()$. As discussed in Section 3.2.2, the time taken to generate the canonical permutation $cannonPID$ corresponding to a permutation PID is $O(n^2)$. Thus the time complexity for each $map()$ function of VFA FEEC computation job is $O(n^2)$. This $map()$ function is called $n!$ times. With M map tasks being executed concurrently, the time complexity for the map phase of the VFA FEEC computation job, when it uses VFA scheme 3, is $O(n^2 \cdot n!/M)$. ■

Proposition 20 *Time complexity for the reduce phase of the VFA FEEC computation job is $O(|K|^n \cdot n_{vfaFE}/R)$, when it uses either VFA scheme 1 or VFA scheme 2. The time complexity for the reduce phase of the VFA FEEC computation job, when it uses VFA scheme 3, is $O((n! + |K|^n \cdot n_{vfaFE})/R)$.*

Proof. Each $reduce()$ function (Algorithm 30) of the VFA FEEC computation job, when using either VFA scheme 1 or VFA scheme 2, takes $O(|K|^n)$ time to generate key-value pairs. This $reduce()$ function is called $O(n_{vfaFE})$ times. With R reduce tasks being executed concurrently, the reduce phase of the VFA FEEC computation job takes $O(|K|^n \cdot n_{vfaFE}/R)$ time.

The time complexity for each $reduce()$ function (Algorithm 34) of the VFA FEEC computation job that uses VFA scheme 3, can be obtained as follows. The list $PIDList$ has $O(n_{vfafePIDs})$ PIDs in it and we iterate over these PIDs to obtain the string $feecPerms$ in $O(n_{vfafePIDs})$ time. Since the total number of system states is $O(|K|^n)$, each $reduce()$ function takes $O(|K|^n)$ time to generate key-value pairs. Thus the time complexity for each $reduce()$ function is $O(n_{vfafePIDs} + |K|^n)$. This function is called $O(n_{vfaFE})$ times. With R reduce tasks being executed concurrently, the time compleity for the reduce phase of the VFA FEEC computation job using VFA scheme 3 is $O((n_{vfafePIDs}+|K|^n) \cdot n_{vfaFE}/R)$, which is $O((n! + |K|^n \cdot n_{vfaFE})/R)$. ■

Proposition 21 *The space complexity for the map phase of the VFA FEEC computation job is $O(n_{vfafePIDs} \cdot M + n! \cdot n_{vfafePIDs})$, when VFA scheme 1 is used, and $O(n_{vfafePIDs} \cdot M + n!)$, when VFA scheme 2 is used. The space complexity for the map phase of the VFA FEEC computation job, that uses VFA scheme 3, is $O(n^2 \cdot M + n!)$.*

Proof. The space complexity for each *map*() function (Algorithms 29 and 31) of the VFA FEEC computation job, that uses either VFA scheme 1 or VFA scheme 2, depends on the space required by subroutine $genFEPerms\_UpdateGraph()$, and data structures $sortedDerPerms$ and $vfaFEECPerms$. As discussed in Section 3.1.2, $genFEPerms\_UpdateGraph()$ has a space complexity of $O(n_{vfafePIDs})$. Also, both $sortedDerPerms$ and $vfaFEECPerms$ require $O(n_{vfafePIDs})$ space to store a set of permutations that belong to the same VFA FEEC. Thus, the space complexity for each *map*() function of the VFA FEEC computation job, that uses either VFA scheme 1 or VFA scheme 2, is $O(n_{vfafePIDs})$. When VFA scheme 1 is used, this *map*() function is called $n!$ times to produce $n!$ key-value pairs. When VFA scheme 2 is used, this *map*() function is called n_{vfaFE} times to produce $O(n_{vfaFE})$ key-value pairs. Each key-value pair requires $O(n_{vfafePIDs})$ space. With M concurrent map tasks, the space complexity for the map phase of the VFA FEEC computation job is $O(n_{vfafePIDs} \cdot M + n! \cdot n_{vfafePIDs})$, when VFA scheme 1 is used, and $O(n_{vfafePIDs} \cdot M + n_{vfaFE} \cdot n_{vfafePIDs})$, when VFA scheme 2 is used. Note that $n_{vfaFE} \cdot n_{vfafePIDs} = O(n!)$.

The space complexity for each *map*() function (Algorithm 33) of the VFA FEEC computation job, using VFA scheme 3, depends on the space complexity of the subroutine $genCannonPerm()$. As discussed in Section 3.2.2, $genCannonPerm()$ has a space complexity of $O(n^2)$. This *map*() function is called $n!$ times, thereby generating $n!$ key-value pairs. Each of these key-value pairs require constant space. With M concurrent map tasks, the space complexity for the map phase of the VFA FEEC computation job is $O(n^2 \cdot M + n!)$. ■

Proposition 22 *The space complexity for the reduce phase of the VFA FEEC computation job is* $O(n_{vfafePIDs} \cdot R + n! \cdot |K|^n)$ *for all of the VFA schemes 1, 2 and 3.*

Proof. Each *reduce*() function (Algorithm 30) of the VFA FEEC computation job, that uses either VFA scheme 1 or VFA scheme 2, requires $O(n_{vfafePIDs})$ space to store $vfaFEECPerms$. This function is called n_{vfaFE} times and each call produces $|K|^n$ key-value pairs. With R concurrent reduce tasks, the total space complexity of the reduce phase of the VFA FEEC computation job is $O(n_{vfafePIDs} \cdot R + n_{vfafePIDs} \cdot n_{vfaFE} \cdot |K|^n)$. Note that $n_{vfaFE} \cdot n_{vfafePIDs} = O(n!)$.

Each *reduce*() function (Algorithm 34) of the VFA FEEC computation job, that uses VFA scheme 3, requires $O(n_{vfafePIDs})$ space to store the string $feecPerms$. Also, each *reduce*() function produces $|K|^n$ key-value pairs, each of which requires $O(n_{vfafePIDs})$ space. With R concurrent reduce tasks and n_{vfaFE} calls to the *reduce*() function, the space complexity for the reduce phase of the VFA FEEC computation job is $O(n_{vfafePIDs} \cdot R + n_{vfafePIDs} \cdot n_{vfaFE} \cdot |K|^n)$. Note that $n_{vfaFE} \cdot n_{vfafePIDs} = O(n!)$. ■

Proposition 23 *Time complexity for the map phase of the system state transition evaluation job is* $O(n_{vfaFE} \cdot |K|^n \cdot n^2/M)$. *Time complexity for the reduce phase of the system state transition evaluation job is* $O(n \cdot |K|^n \cdot log(|K|) \cdot n_{vfaFE}/R)$.

Proof. The time complexities for each *map*() (Algorithm 35) and *reduce*() function (Algorithm 36) of system state transition evaluation job in non-iterative VFA GDS evaluation paradigm is the same as that of each *map*() and *reduce*() function of the same job used in non-iterative brute force GDS evaluation paradigm. As discussed in Section B.2.2.2, each *map*() and *reduce*() function take $O(n^2)$ and $O(|K|^n \cdot log(|K|^n))$ time, respectively, to execute. A *map*() function is called $O(n_{vfaFE} \cdot |K|^n)$ times; while a *reduce*() function is called $O(n_{vfaFE})$ times. With M map tasks being executed concurrently, the time complexity for the map phase of the system state transition evaluation job is $O(n_{vfaFE} \cdot |K|^n \cdot n^2/M)$. With R concurrent reduce tasks, the time complexity for the reduce phase of the system state transition evaluation job is $O(|K|^n \cdot log(|K|^n) \cdot n_{vfaFE}/R)$. ■

Proposition 24 *The space complexity for the map phase of the system state transition evaluation job is* $O((n^2+n_{vfafePIDs}) \cdot M + n! \cdot |K|^n)$ *and that of the reduce phase is* $O((|K|^n + n_{vfafePIDs}) \cdot R + n_{vfaFE} \cdot |K|^n + n!)$.

Proof. The space complexity for each *map*() function of system state transition evaluation job in non-iterative VFA GDS evaluation paradigm is equal to the sum of the storage space (as discussed in Section B.2.2.2) required by each *map*() function of the same job used in non-iterative brute force GDS evaluation paradigm, and the space required to store $vfaFEECPerms$. $vfaFEECPerms$ requires $O(n_{vfafePIDs})$ storage space. Thus the space complexity for each *map*() function is $O(n^2 + n_{vfafePIDs})$. This *map*() function is called $O(n_{vfaFE} \cdot |K|^n)$ times in the map phase, thereby producing $O(n_{vfaFE} \cdot |K|^n)$ key-value pairs, each of which requires $O(n_{vfafePIDs})$ space. Therefore, the total space complexity of the map task is $O((n^2 + n_{vfafePIDs}) \cdot M + n_{vfaFE} \cdot n_{vfafePIDs} \cdot |K|^n)$, which is $O((n^2 + n_{vfafePIDs}) \cdot M + n! \cdot |K|^n)$.

Also, the space complexity for each *reduce*() function of system state transition evaluation job in non-iterative VFA GDS evaluation paradigm is equal to the sum of the space complexity (as discussed in Section B.2.2.2) for each *reduce*() function of the same job used in non-iterative brute force GDS evaluation paradigm, and the space required to store $fePIDs$. Since $fePIDs$ requires $O(n_{vfafePIDs})$ space, the space complexity for each *reduce*() function is $O(|K|^n + n_{vfafePIDs})$. Each *reduce*() function is called $O(n_{vfaFE})$ times. This produces $O(n_{vfaFE})$ key-value pairs, each of which requires $O(|K|^n + n_{vfafePIDs})$ storage space. Thus the space required to store all key-value pairs generated by the reduce phase is $O(n_{vfaFE} \cdot (|K|^n + n_{vfafePIDs}))$ and the space complexity of the reduce phase is $O((|K|^n + n_{vfafePIDs}) \cdot R + n_{vfaFE} \cdot (|K|^n + n_{vfafePIDs}))$. Note that $n_{vfaFE} \cdot n_{vfafePIDs} = O(n!)$. ■