SPECIAL ISSUE ARTICLE

Optimizing Concurrency under Scheduling by Edge Reversal

Carlos E. Marciano[1] | Gladstone M. Arantes Jr[1,2] | Abilio Lucena[1] | Felipe M. G. França[1] | Luerbio Faria[3] | Luidi G. Simonetti[1]

[1]Systems Engineering and Computer Science Program, Federal University of Rio de Janeiro, RJ, Brazil

[2]BNDES – Brazilian Development Bank, Rio de Janeiro, RJ, Brazil

[3]Institute of Mathematics and Statistics, Rio de Janeiro State University, RJ, Brazil

Correspondence
Carlos E. Marciano, Systems Engineering and Computer Science Program, Federal University of Rio de Janeiro, RJ, Brazil
Email: cemarciano@poli.ufrj.br

Funding information
CNPq, FAPERJ, CAPES

Scheduling by Edge Reversal provides an order of operation for nodes in a graph, but maximizing or minimizing the resulting concurrency is hard. In this paper, we discuss a series of real-world applications for this technique and propose algorithms for both problems. For maximum concurrency, we prove an approximability result and introduce an approximation algorithm. For minimum concurrency, we provide a technique to solve this problem to proven optimality, and introduce a novel application for assembling *musical phrases*. This paper is an extended version of a conference paper.

KEYWORDS
concurrency, heavily loaded systems, optimization, computer music

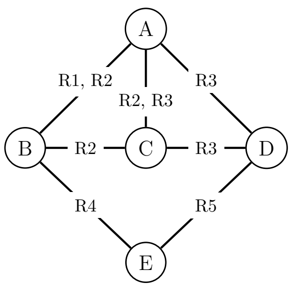
1 | INTRODUCTION

Challenges in distributed systems permeate the modern world. Seemingly distinct applications such as process scheduling on a CPU, traffic control on road junctions and firefighting strategies with autonomous robots have at least one common constraint: limited resources. This simple yet powerful characteristic is enough to allow resource-sharing algorithms to be used broadly in a variety of domains, providing reliable solutions with mathematical guarantees that are often straightforward to implement. However, resource allocation alone is of little use if not done efficiently: in fact, despite intuition, it is natural to even ask how to measure the efficiency of a distributed system. And, even more natural, to ask how to optimize it.

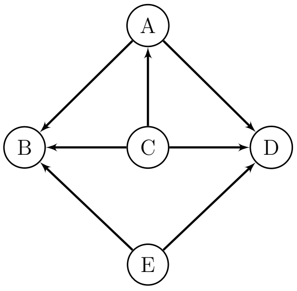
In an attempt to answer some of these fundamental questions, Barbosa and Gafni [1] introduced, in 1989, a distributed algorithm known as *Scheduling by Edge Reversal (SER)*. Under a heavy-load assumption, where each participant (or process) actively requires all of its corresponding resources to perform an action and *operate*, *SER* is able to model resource-sharing systems as unweighted, connected graphs. Specifically, a process that takes part in the distributed dynamic can be represented as a node in the graph, while an edge between any two nodes symbolizes shared resources between them. As a result, multiple nodes requiring the same resource form a *clique*, a complete sub-graph where each node will only be able to *operate* when all other processes in the *clique* are *idle*. Figure 1(a) illustrates this particular network, which shall be referred to as "resource graph" throughout this paper.

The execution of *SER* is fairly intuitive: once a *resource graph* has been modeled from a real-world problem, we can apply any initial acyclic orientation to the graph (Figure 1(b)) and start the edge-reversal dynamics. At each time step, *SER* simultaneously *operates* all *sinks*, meaning that every node with no outgoing edges is able to utilize its corresponding resources and perform its given task. Once all *sink nodes* are done *operating*, they simultaneously revert all their edges, essentially creating new *sinks* throughout the graph and bringing the system to a new acyclic orientation. Although this process is repeated indefinitely, the succession of distinct orientations induced by this dynamic is finite, leading to a recurring sequence of acyclic orientations that is often referred to as a *period*. Moreover, as demonstrated by Barbosa and Gafni [1], all nodes in the *resource graph operate* the same number of times within a *period*. Figure 1(c) illustrates a *period* of length 3, where each node *operates* once.

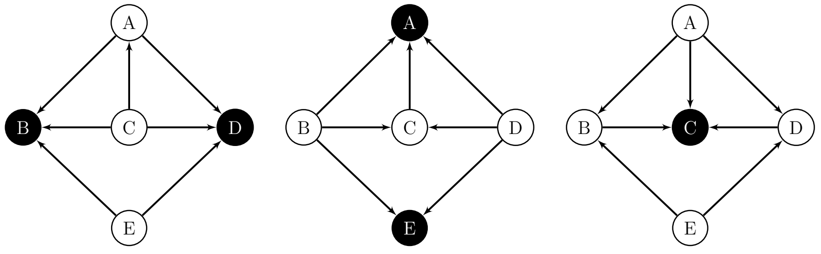
Although a formal metric for concurrency will only be defined in Section 2, it is already possible to visualize that distinct initial orientations will result in more or less nodes *operating* simultaneously, providing varying degrees of



(a) A *resource graph*. Nodes sharing resources are connected by an edge.



(b) A given acyclic orientation applied to the *resource graph* from (a).



(c) A schedule (from left to right) taking (b) as the initial orientation. Sinks (in black) are allowed to *operate*, after which they revert their edges so new sinks may be formed. Once node C is done *operating*, the leftmost orientation will be repeated, leading to a period.

FIGURE 1 Scheduling by Edge Reversal as a distributed solution for scheduling processes (nodes) in a resource-sharing system.

concurrency depending solely on how a given system is initialized. As a consequence, it is only natural to ask how to obtain the acyclic orientations that optimize this value of concurrency, which can be a minimization problem for some applications or, more commonly, a maximization objective. In fact, the decision problems associated with obtaining minimum and maximum concurrency have been proved to be NP-complete by Arantes Jr [2] and by Barbosa and Gafni [1], respectively, and much still needs to done to fully understand and model these optimization problems.

In an attempt to bridge this gap, our paper studies both problems from a theoretical perspective, ultimately proposing different algorithmic approaches for each of these optimization objectives. Initially, we review the literature surrounding *SER* in Section 2, introducing the relevant notation and analyzing different applications proposed throughout the years that are relevant to the two sections that follow. First, Section 3 is dedicated to the problem of maximizing concurrency, presenting a negative result for approximability as well as a $2/\Delta$–approximation algorithm for graphs with maximum degree Δ. In turn, Section 4 studies the problem of minimizing concurrency, introducing a computational model for this task as well as an original application of *SER* for assembling a maximum-length loop of computer music. Finally, our concluding remarks are expressed in Section 5, alongside suggestions for future work.

2 | LITERATURE REVIEW

Having described the operation mechanism of *SER* in Section 1, we dedicate this section to a brief literature review that builds the foundation for the remainder of this paper. First, in Subsection 2.1, we introduce the necessary notation and definitions that will be referenced in later sections. In turn, in Subsection 2.2, we highlight some of the most prominent applications involving *SER*, given that no summary of *SER* applications currently exists in the literature.

2.1 | Notation and Definitions

We first recall all the necessary terminology presented in Barbosa and Gafni [1] to define concurrency under *SER*. Moreover, we note that two equivalent expressions for concurrency are known, both of which are a function of an acyclic orientation ω. The first, which we refer to as a *dynamic definition*, is a simple equation calculable in polynomial time that requires a full execution of the edge-reversal dynamics in order to be determined. The second, which we call a *static definition*, is a combinatorial expression, and is largely used in Section 4 due to its optimization nature. We stress that both functions produce the same numerical result, since the domain, the image and the resulting association are the same.

Let $G = (V, E)$ be a connected undirected graph such that $|E| \geq |V|$ (that is, G is not a tree). We first formally define an **acyclic orientation** of G as a function $\omega : E \to V$, which takes an edge $e \in E$ as a parameter and outputs the vertex to which e is oriented towards. In order for this orientation to be acyclic, given any undirected cycle of the form $i_0, i_1, ..., i_{|\kappa|-1}, i_0$, the function ω should never return $\omega(i_0, i_1) = i_1, \omega(i_1, i_2) = i_2, ..., \omega(i_{|\kappa|-1}, i_0) = i_0$. We also define the set of all acyclic orientations of G as Ω.

Moreover, we define a ***period*** of length p as a sequence of acyclic orientations $\omega_0, ..., \omega_{p-1}$ which is periodically repeated when executing *SER*. Additionally, let m be the number of times that each node *operates* within a *period*, which is equal to all nodes. The following function $\gamma : \Omega \to \mathbb{R}$ is a *dynamic definition* of concurrency, and we invite the reader to verify that it evaluates to $1/3$ for the edge-reversal dynamic in Figure 1(c):

$$\gamma(\omega) = \frac{m}{p} \tag{1}$$

We now focus our attention to the *static definition* of concurrency, for which a few other definitions are necessary. First, we denote by $\kappa \subseteq V$ an **undirected simple cycle** of G, that is, a set of vertices that constitute a sequence of length $|\kappa| + 1$ of the form $i_0, i_1, \ldots, i_{|\kappa|-1}, i_0$. It is also possible to arbitrarily define directions of traversal: if the vertices of κ are visited from i_0 to $i_{|\kappa|-1}$, we say that κ is traversed in the clockwise direction. Otherwise, we say that κ is traversed in the counterclockwise direction. For combinatorial reasons, we shall denote by uppercase K the set of all simple cycles of G.

At this point, we need one final definition before we can formalize the notion of *static concurrency*. As such, given an undirected simple cycle κ and an acyclic orientation ω, we denote by $n_{cw}(\kappa, \omega)$ the number of edges in κ oriented by ω in the clockwise direction. Likewise, we denote by $n_{ccw}(\kappa, \omega)$ the number of edges oriented in the counterclockwise direction. Finally, let the concurrency of a graph G be defined as a function $\gamma : \Omega \rightarrow \mathbb{R}$ such that:

$$\gamma(\omega) = \min_{\kappa \in K} \left\{ \frac{\min \{n_{cw}(\kappa, \omega), n_{ccw}(\kappa, \omega)\}}{|\kappa|} \right\} \tag{2}$$

In other words, for every simple cycle κ of G, we calculate the number of edges oriented in the clockwise direction as well as the ones in the counterclockwise direction. We take the minimum of these two values and divide it by the size of the undirected cycle κ. Whichever $\kappa \in K$ produces the smallest value will dictate the concurrency of the system.

2.2 | *SER* Applications

Over the years, *SER* has inspired a number of inventive solutions spanning different real-world applications. In this subsection, we discuss a few of these contributions, highlighting whether they benefit more from maximum or minimum concurrency. Regardless, the algorithmic procedures presented later in Section 3 and Section 4 should help provide, to different extents, better solutions to all of these applications.

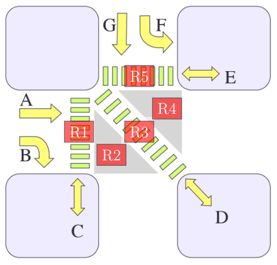
2.2.1 | Scheduling Processes on a CPU

One immediate example that clearly benefits from maximum concurrency is process scheduling. Barbosa and Gafni [1], inspired by *Dijkstra's Five Dining Philosophers* [3, 4], exemplified this application when describing *SER*. In short, five philosophers, each serving as an analogy for a process, sit at a round table in front of meals, which are separated by forks. For this specific variation, each philosopher only has two states: *eating*, where they consume both forks closest to them and dine; or *hungry*, in which they anxiously wait until the two forks in close proximity are available.

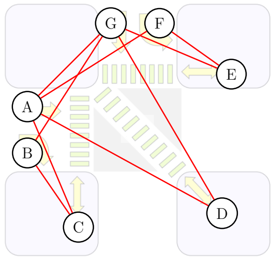
This lighthearted anecdote illustrates many fundamental concepts that need to be taken into account when designing operating systems [4]. Particularly, *SER* is capable of preventing *deadlocks*, since all orientations are acyclic; and able to prevent *starvation*, as each process is given an equal amount of time to perform its task. In fact, since all processes *operate* the same number of times within a period, and since a given node will only *operate* again after all its neighbors have also *operated*, we say that *SER* is able to guarantee *fairness* [1].

2.2.2 | Traffic Control in Road Junctions

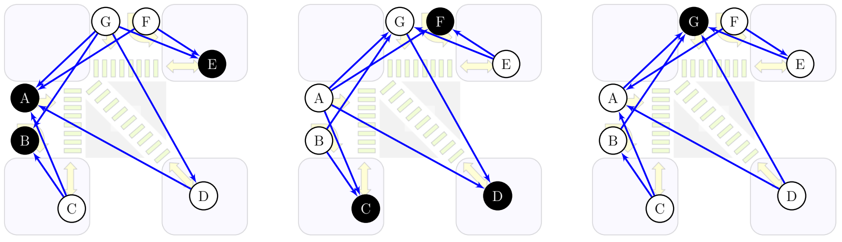
When introducing a variation of *SER* for systems in near-heavy load, Carvalho *et al.* [5] proposed an application of the original algorithm for road intersections, scheduling the different flows of vehicles and pedestrians. Similarly, for the first time in a publication, we illustrate this technique using the *Shibuya Crossing* in Tokyo, which is one of the busiest and



(a) Diagram of the *Shibuya Crossing* with resource indications.



(b) The *resource graph* associated with the *Shibuya Crossing*.



(c) A schedule (from left to right) originated from an arbitrary acyclic orientation ω that leads to a *period* of length 3 and a value of concurrency $\gamma(\omega) = 1/3$.

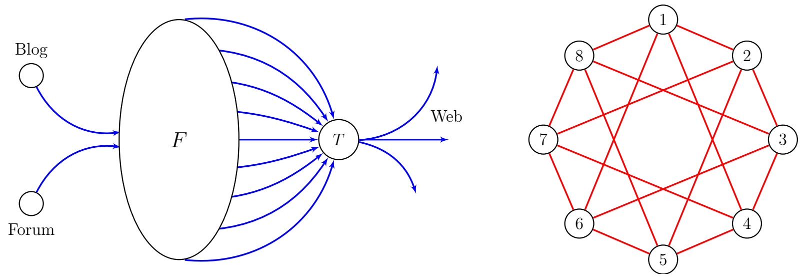
FIGURE 2 Scheduling by Edge Reversal applied to the *Shibuya Crossing* in *Tokyo*.

most famous road intersections in the world. We start by presenting a diagram of the crossing in Figure 2(a), including indications of how each area will be encoded as a resource.

Having identified which physical areas are shared by each flow of vehicles and pedestrians (labeled from A to G), we are able to build the corresponding *resource graph* shown in Figure 2(b). For instance, flows A, D and G require the central lane $R3$ in order to *operate*, which is why they form a *clique* in the *resource graph*. Ultimately, we apply a desired acyclic orientation to the graph and execute the edge-reversal dynamics, as shown in Figure 2(c): whenever a node *operates*, it means that the flow it represents is allowed to take place. As long as the *resource graph* was correctly modeled, no collision caused by the schedule will ever occur.

2.2.3 | Decontamination of WebGraphs

One common practice to increase the visibility of a given *Web* page T when performing a *Web* search is to inject artificial links to T on a variety of *websites*. As such, malicious users can create a *link farm* F, comprised of various pages that point towards T, and spread links to F on *forums*, *blogs*, or other social networks. This is illustrated in Figure 3(a), where the *link farm* F is represented by an ellipse that denotes a dense graph. In order to model this densely connected *link farm* F, some researchers resort to *circulant graphs* [6], where each vertex i is connected to vertices $i + j$ and $i - j$ for each j in a list L. As such, these graphs are represented as by the prefix "*Ci*", followed by the number of vertices n, and a list with the values of j. For instance, $Ci_n(1, 2, ..., \lfloor n/2 \rfloor)$ denotes the complete graph K_n, while $Ci_n(1)$ represents the circular graph C_n. We illustrate this class of graphs in Figure 3(b).



(a) Diagram of a *link farm* F. (b) Graph $Ci_8(1, 3)$.

FIGURE 3 Possible layout of a malicious network and its *link farm* as a circulant graph.

In order to weaken this malicious network, a decontamination strategy for the *link farm F* is employed, where *Web Marshalls (WMs)* traverse each node and erase unwanted links. However, recontamination is possible, so a *WM* may only leave a node after most (or, in some cases, all) of its neighbors have also been *guarded* with other *WMs*. The problem of minimizing the number of *WMs* (or, more generally, any other agent) is known as the *node searching problem* [7], and Gonçalves*et al.* [8] have proposed an algorithm using *SER* to obtain a decontamination strategy for the *link farm F* that provides a good solution despite not being optimal. In short, at a given time step, this algorithm: sends replicated *Web Marshals* to nodes that will become *sinks* in the next iteration; checks if *WMs* can be terminated; and applies the edge-reversal dynamics so that these recently created *WMs* may replicate themselves to new nodes. As a consequence, high values of concurrency imply in an excessive number of *WMs* being employed, so Gonçalves*et al.* [8] have noted that minimum concurrency is desirable for this application, despite not providing a formal relation between the value of concurrency and the number of *WMs*.

2.2.4 | Firefighting Scenarios

Alves *et al.* [9] have extended the decontamination strategy presented in Subsection 2.2.3 to firefighting scenarios in apartments, where *agents* are autonomous robots and nodes correspond to fire outbreaks. This makes the problem more challenging to model due to physical constraints, requiring an additional graph to encode how robots may move from one room to another. Moreover, the authors have simulated this strategy both in a virtual environment as well as in a real-life architecture model, using miniature robots to mimic the autonomous firefighters.

2.2.5 | Job Shop Applications

Typically, in problems involving job shops, a set of jobs, each consisting of a series of tasks that must be executed in order, are to be scheduled to a set of machines. The objective is to minimize the time of conclusion associated with the very last job, which is NP-hard even with just 3 jobs and 3 machines [10]. In order to assign each job to a given machine, Lengerke *et al.* [11] proposed a variation of *SER* to obtain viable solutions in which sinks don't perform the reversal of edges simultaneously, but instead when the task they represent is completed. Additionally, they have employed *SER* to plan routes for *Automated Guided Vehicles (AGVs)*, which are responsible for transporting materials between each machine, such that no collision occurs. This idea is similar to the traffic control application presented in Subsection 2.2.2, but has additional constraints due to the scheduling of jobs.

TABLE 1 Decision and optimization problems mentioned throughout this paper.

CONCUR **(decision)**
INSTANCE: Graph $G = (V, E)$ and positive integers m and p.
QUESTION: Is there an acyclic orientation $\omega \in \Omega$ with concurrency $\gamma(\omega) \geq \frac{m}{p}$?
MAXCON — MAXIMUM CONCURRENCY **(optimization)**
INSTANCE: Graph $G = (V, E)$.
GOAL: Find an acyclic orientation $\omega \in \Omega$ which maximizes the concurrency $\gamma(\omega) = \frac{m}{p}$.
MIN^{-1}CON — MINIMUM OF THE INVERSE OF THE CONCURRENCY **(optimization)**
INSTANCE: Graph $G = (V, E)$.
GOAL: Find an acyclic orientation $\omega \in \Omega$ which minimizes the value $\frac{p}{m}$.
MINCON — MINIMUM CONCURRENCY **(optimization)**
INSTANCE: Graph $G = (V, E)$.
GOAL: Find an acyclic orientation $\omega \in \Omega$ which minimizes the concurrency $\gamma(\omega) = \frac{m}{p}$.

3 | MAXIMUM CONCURRENCY RESULTS

In this section, we study the problem of finding an acyclic orientation for a given resource graph that leads to the maximum possible concurrency under *Scheduling by Edge Reversal (SER)*. All the decision and optimization problems mentioned throughout this section are presented in Table 1. We discuss our first result in Subsection 3.1, where we use an *L-reduction* [12] to show that CONCUR is not approximable with a ratio less than $n^{\frac{1}{7}-\epsilon}$, extending the result presented in Bellare *et al.* [13] for MINIMUM GRAPH COLORING. In turn, in Subsection 3.2, we introduce the first known approximation algorithm for maximizing concurrency in connected graphs with maximum degree Δ, which outputs an acyclic orientation leading to concurrency at most $2/\Delta$.

3.1 | Approximability

Given a graph $G = (V, E)$, Bellare *et al.* [13] proved that, unless $P = NP$, COLORING cannot be approximated in a ratio less than $n^{\frac{1}{7}-\epsilon}$, for every $\epsilon > 0$. In Theorem 1, we extend this result to CONCUR. We note that we had to introduce the problem MIN^{-1}CON because COLORING is a minimization objective. However, MIN^{-1}CON is pragmatically equivalent to MAXCON.

Theorem 1 *Given a graph $G = (V, E)$ with n vertices, then, unless $P = NP$,* MIN^{-1}CON *cannot be approximated in a ratio less than $n^{\frac{1}{7}-\epsilon}$, for every $\epsilon > 0$.*

Proof It is enough to present an *L-reduction* [12] from COLORING to MIN^{-1}CON, because if a problem P_1 L-reduces to a problem P_2 and the problem P_2 has a polynomial r-approximation algorithm, we conclude that, up to the constants, problem P_1 also has a polynomial r-approximation algorithm. Let $\chi(G)$ denote the *chromatic number* of G. As such, in order to L-reduce COLORING to MIN^{-1}CON, one must yield f and g, a pair of polynomial time algorithms in the size of G, and a pair of positive reals α and β, such that, given an instance $G = (V, E)$ of COLORING, algorithm f produces $f(G) = H = (V_H, E_H)$, an instance of MIN^{-1}CON, satisfying:

(i) $Opt_{\text{MIN}^{-1}\text{CON}}(H) \;\leq\; \alpha \, Opt_{\text{COLORING}}(G) \;=\; \alpha \, \chi(G)\,;$

(ii) given a feasible solution η_H for MIN^{-1}CON in H, algorithm g obtains a feasible solution ξ_G for COLORING in G such that:

$$\left|\chi(G) - \xi_G\right| \;\leq\; \beta \left|Opt_{\text{MIN}^{-1}\text{CON}}(H) - \eta_H\right| \tag{3}$$

Algorithm f, which yields the instance $H = f(G)$ of MIN^{-1}CON, is defined by adding a *universal vertex* v to G, $v \notin V(G)$, and for all $u \in V(G)$, $vu \in E(H)$, i.e., $V(H) = V(G) \cup \{v\}$, and $E(H) = E(G) \cup \{vu : u \in v(G)\}$. Given a coloring of G with colors $1, 2, 3, \ldots, \chi(G)$, one can extend this coloring to H by assigning an extra color $\chi + 1$ to v. Next, consider an orientation ω for the edges of G by setting $\omega(uw) = w$ if:

1. u has color c_u;
2. w has color c_w;
3. and $c_w < c_u$.

At some point, the *universal vertex* v will be the only sink during a particular acyclic orientation within a *period*, and the directed edges from all vertices of G to v will force each set of sinks with the colors $1, 2, 3, \ldots, \chi$ to *operate* in sequence following the *operation* of v. This defines a period of length $p = \chi + 1$. Hence:

$$Opt_{\text{MIN}^{-1}\text{CON}}(H) \;\leq\; p \;=\; \chi + 1 \;\leq\; 2\chi \;=\; 2\, Opt_{\text{COLORING}}(G) \tag{4}$$

Note that $\alpha = 2$ suffices, and this allows us to conclude the first part of the L-reduction.

Additionally, if η_H is a solution for MIN^{-1}CON in H with cost p/m, then necessarily $m = 1$. Given that no other node will *operate* alongside the *universal vertex* v, then the vertices in $V(H) \setminus \{v\}$ will become sinks in sequence until v becomes a sink again, with no vertex ever operating twice before v *operates* again. Hence, the sinks of $H \setminus \{v\}$ define a partition of $V(G)$ into $p - 1$ independent sets, and this is the definition of algorithm g. As a result, we point out that, in Equation 3, the assignment $\beta = 1$ suffices:

$$\left|\chi(G) \;-\; (p-1)\right| \;\leq\; \left|\chi(G) + 1 - p\right| \;=\; \left|Opt_{\text{MIN}^{-1}\text{CON}}(H) \;-\; p\right| \tag{5}$$

This concludes the *L-reduction*. □

Algorithm 1: Approximation algorithm for finding an acyclic orientation that leads to concurrency at most $2/\Delta$.

```
Input  : Graph $G = (V, E)$ with maximum degree $\Delta$, where $G$ is
         neither a complete graph nor an odd cycle on $n$ vertices
Output: Acyclic orientation $\omega_\Delta$ associated to concurrency at most $2/\Delta$
Run Lovász's algorithm, obtaining a partition of $V$ into the independent
  sets $(V_1, V_2, V_3, \ldots, V_\Delta)$
Create an empty orientation $\omega_\Delta$
foreach edge $uv \in E$, where $u \in V_i$ and $v \in V_j$ do
    if $i < j$ then
        Orient edge such that $\omega_\Delta(u, v) = u$
    end
    else
        Orient edge such that $\omega_\Delta(u, v) = v$
    end
end
return $\omega_\Delta$
```

3.2 | Approximation Algorithm

In this subsection, we introduce an approximation algorithm for graphs with maximum degree Δ. In 1941, Brooks [14] proved that, if a graph $G = (V, E)$ has maximum degree Δ; chromatic number χ; is connected; and is neither an odd cycle nor a complete graph, then $\chi \leq \Delta$. Later, in 1975, Lovász [15] exhibited a polynomial algorithm which obtains a Δ-coloring of G. The algorithmic procedure introduced in this subsection is strongly based on Lovász's algorithm, and we structure it in Algorithm 1.

Lemma 1 *Let $G = (V, E)$ be a connected graph with maximum degree Δ, where G is neither an odd cycle nor a complete graph, and let ω_Δ, the output of Algorithm 1, be an acyclic orientation over E with concurrency $\gamma(\omega_\Delta) = \frac{m}{p}$. Then, we have that $\frac{1}{\Delta} \leq \frac{m}{p} \leq \frac{1}{2}$.*

Proof Initially, we shall prove the leftmost inequality. Note that the transformation of sink vertices into source vertices of an acyclic orientation does not increase the longest oriented path of the graph. Note also that an oriented path in ω_Δ has size at most Δ. Then, the maximum number of consecutive steps within a *period* in which a vertex is not a sink is Δ. Hence, after $m\Delta$ steps, all vertices of G will have operated at least m times, and so the p orientations of the *period* must have been concluded, i.e., $m\Delta \geq p$. The second inequality is proved by the observation that a vertex cannot operate twice within a period of length p. Hence, $2m \leq p$. □

Theorem 2 *The performance ratio of Algorithm 1 is at most $2/\Delta$.*

Proof By Lemma 1, one has that $\frac{1}{\Delta} \leq \gamma(\omega_\Delta) \leq \frac{1}{2}$. Hence, the performance ratio of Algorithm 1 is bounded by:

$$R_A = \frac{|\omega_\Delta|}{|Opt_{\text{MIN}^{-1}\text{CON}}(G)|} \geq \frac{1/\Delta}{1/2} = \frac{2}{\Delta} \tag{6}$$

□

4 | MINIMUM CONCURRENCY RESULTS

In this section, we expand on the contributions associated with minimum concurrency under *SER* that were first introduced in the conference version of this paper [16]. In Subsection 4.1, we discuss a computational approach for solving the *minimum concurrency problem* to proven optimality and provide experimental results for a variety of random graphs, showing in Subsection 4.2 that our strategy is capable of dealing with fairly large instances. Additionally, in Subsection 4.3, we present a musical application involving *minimum concurrency* that is capable of assembling pre-recorded musical *phrases* into maximum-length loops of original computer tracks, while also respecting fundamental concepts in music theory. Finally, in this same section, we present an interactive simulation showcasing the aforementioned strategy, which is available on *Web browsers* alongside its *open-source* code.

4.1 | Deterministic Model

Before the development of the computational approach presented in this subsection, no strategies for minimizing concurrency to optimality existed. In fact, instead of building an overly complicated model for this problem specifically, we resort to the well-known *simple cycle problem* and describe a linear-time algorithm that provides an optimal solution to MINCON given a *longest simple cycle* as an input. This allows us to make use of the well-established literature for finding maximum cycles, which is NP-complete [17] but features a number of optimization models that were perfected over the last decades [18].

Initially, we shall derive an expression for obtaining the minimum value of concurrency. As such, given that Equation 2 is a function of an acyclic orientation ω, we minimize this relation over all $\omega \in \Omega$ and assign it to a value γ^*:

$$\gamma^* = \underset{\omega \in \Omega}{min} \left\{ \underset{\kappa \in K}{min} \left\{ \frac{min\ \{n_{cw}(\kappa, \omega), n_{ccw}(\kappa, \omega)\}}{|\kappa|} \right\} \right\} \tag{7}$$

In its current state, γ^* contains three nested minimization expressions, two of which are over exponentially large sets, which can prove to be an excessive challenge to calculate. In this regard, in order to significantly simplify this equation, we present the lemma below:

Lemma 2 $\gamma^* = \underset{\kappa \in K}{min} \left\{ \frac{1}{|\kappa|} \right\}$.

Proof Consider Equation 2, the *static definition* of concurrency. For a given ω', let κ' be the simple cycle that minimizes the internal fraction. Let x be defined as the numerator $x = min\ \{n_{cw}(\kappa', \omega'), n_{ccw}(\kappa', \omega')\}$, which brings Equation 2 to a value of $\gamma(\omega') = x/|\kappa'|$.

However, for every $\kappa \in K$, there will always exist an acyclic orientation ω such that $n_{cw}(\kappa, \omega) = 1$ and $n_{ccw}(\kappa, \omega) = |\kappa| - 1$, or vice versa. This follows immediately from the fact that a directed cycle would only exist if and only if either $n_{cw}(\kappa, \omega) = 0$ or $n_{ccw}(\kappa, \omega) = 0$.

Therefore, there must also exist an orientation ω for κ' such that either $n_{cw}(\kappa', \omega) = 1$ or $n_{ccw}(\kappa', \omega) = 1$. Consequently, if ω', when applied to κ', didn't produce the result $x = 1$, there will necessarily exist another acyclic orientation ω that will lead to $\gamma(\omega) = 1/|\kappa'|$.

Now, consider Equation 7. If γ^* is less than $1/|\kappa'|$, then there must exist a simple cycle κ^* which, under an orientation ω^*, will produce $1/|\kappa^*| < 1/|\kappa'|$. As such, Equation 7 has become a minimization problem over all $\kappa \in K$. □

Algorithm 2: A linear-time algorithm for finding an acyclic orientation that leads to minimum concurrency given a longest cycle as input.

```
Input  : Undirected graph $G = (V, E)$ and longest cycle $\kappa^* \subseteq V$
Output: Acyclic orientation $\omega^*$ for which $\gamma(\omega^*)$ is minimum
id = 1
v = $\kappa^*$.getFirstVertex()
for i=1 to $\kappa^*$.size() do
    Assign id to v
    Increment id
    v = $\kappa^*$.getClockwiseNeighborOf(v)
end
while a vertex $v \in V$ with no id exists do
    Assign id to v
    Increment id
end
Create an empty orientation $\omega^*$
foreach undirected edge $uv \in E$ do
    if $id(v) > id(u)$ then
        Orient edge such that $\omega^*(u, v) = u$
    end
    else
        Orient edge such that $\omega^*(u, v) = v$
    end
end
return $\omega^*$
```

Lemma 2 states that, in order to calculate the minimum possible concurrency of a given *resource graph G*, one can instead look for a longest simple cycle of G (which is NP-complete [17]) and simply calculate the reciprocal of its cardinality. Therefore, many tools previously available in the literature for identifying longest cycles may now be used to obtain the value of minimum concurrency. It is still necessary, however, to find an acyclic orientation ω^* that will lead the system to a schedule corresponding to minimum concurrency. In this context, we present the following theorem:

Theorem 3 *Given any longest cycle $\kappa^* \in K$ as input, there exists a linear-time algorithm for finding an orientation $\omega^* \in \Omega$ such that $\gamma(\omega^*)$ is minimum over all $\omega \in \Omega$.*

Proof The proof of Lemma 2 states that minimum concurrency will be attained if an orientation ω^* is applied to G under the condition that $n_{cw}(\kappa^*, \omega^*) = 1$ and $n_{ccw}(\kappa^*, \omega^*) = |\kappa^*| - 1$ or vice versa, where κ^* is a longest cycle. Orienting κ^* under the aforementioned conditions can be performed in linear time by traversing the cycle κ^* and assigning an increasing identification number $1, \ldots, |\kappa^*|$ to each visited vertex, resulting in a topological ordering of the cycle. By orienting the corresponding edges towards the vertices with lower identification numbers, only one edge (connecting the vertices with the highest and the lowest identification numbers) will be oriented in the opposite direction from the other $|\kappa^*| - 1$ edges, fulfilling the requirement.

It is now necessary to prove that it is possible to orient the remaining edges of G such that the resulting orientation ω^* is always acyclic. Let $S = V - \kappa^*$ be the set of the remaining vertices of G. Let us assign an increasing identification number $|\kappa^*| + 1, \ldots, |V|$ to each vertex in S, and then orient all edges of G towards the vertices with lower identification numbers. By contradiction, if the resulting orientation ω^* were cyclic, there would need to exist a path $i_0, i_1, \dots, i_0$ (i.e. a directed cycle). However, since edges always lead to vertices of lower identification numbers, it is impossible to return to i_0 after leaving it, for any $i_0 \in V$. As such, no cycles are formed. □

Finally, we formalize the procedure discussed during the proof of Theorem 3 in the form of Algorithm 2, and note that its correctness relies on the aforementioned proof. Additionally, in order for Algorithm 2 to attain linear time, the method *getClockwiseNeighborOf(v)* must be performed in $O(1)$. This is possible by using a suitable data structure to store κ^*, which is usually an array containing the vertices of the cycle in the order they should appear. In this scenario, *getClockwiseNeighborOf(v)* would simply return the next vertex in the array. We also point out that, since G is always a connected graph where $|E| \geq |V|$, Algorithm 2 has an overall time complexity of $O(m)$, where $m = |E|$.

4.2 | Computational Results

In order to assess the viability of our proposed strategy, we created computational experiments to verify the amount of time needed to solve certain instances of random graphs. Based on the discussions presented in Subsection 4.1, our approach consists of two steps: first, we employ a *branch-and-cut* strategy for finding a longest cycle κ^* of the *resource graph* G; then, we apply Algorithm 2 to obtain, in linear time, an acyclic orientation of G that leads the system to minimum concurrency. The pipeline in Figure 4 summarizes this process.

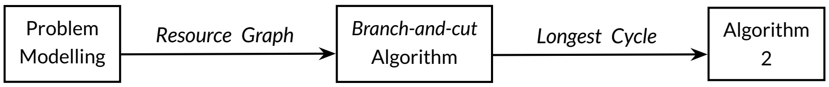


FIGURE 4 Implementation pipeline for obtaining an acyclic orientation that leads to minimum concurrency.

Specifically, in order to solve the *branch-and-cut* step, we employed the model presented in Lucena, Cunha and Simonetti [18] for the *simple cycle problem*. Instead of resorting to cutset inequalities, their algorithm separates *Generalized Subtour Elimination Constraints on the fly* (as they become violated at an LP relaxation in the branch-and-cut enumeration tree). In order to evaluate graphs of varying densities, all of our instances are $G(n, p)$ models [19], which is a graph of n nodes where an edge exists between any two nodes with probability p. For each combination of n and p, we generated 10 different instances with these same properties.

Lastly, the experimental results are collected in Table 2, and all technical details are specified in its caption. Through these experiments, it is possible to see that some random graphs with as many as 2 000 nodes and 40 000 edges can be solved in under one hour, while less dense instances are solved much faster. Increasing the number of nodes and edges beyond this value is challenging, but additional features of the solver that were not used in these experiments could help push this boundary further.

TABLE 2 Computational results for the Simple Cycle Problem model [18] using the *XPRESS Mixed Integer Programming* package v8.5.3 with all other features disabled (pre-processing, primal heuristics, etc). Intel Core i9-8950HK, 16 Gbytes of RAM, Linux Ubuntu 18.04.1, one thread. Execution time limited to 3600 seconds for each of the 10 instances.

Nodes	p	Avg. Edges	Solved	Avg. Min. Conc.	CPU Time (s)
200	0.01	391	10	1 / 178	0.6 (± 0.9)
200	0.1	3 780	10	1 / 200	6.5 (± 7.3)
1000	0.002	2 062	10	1 / 905	73.2 (± 51.4)
1000	0.02	19 695	10	1 / 1000	797.0 (± 547.3)
1000	0.2	179 806	3	1 / 1000	2 619.9 (± 1 015.0)
2000	0.001	4 091	10	1 / 1805	425.9 (± 371.3)
2000	0.01	39 807	3	1 / 2000	2 107.9 (± 1 561.5)
2000	0.1	380 199	0	—	—

4.3 | Musical Application

Previous studies have noted that effective music generation is the dream of computer music researchers [20]. In short, it is possible to separate this field into two main categories, based on the method used to generate the music: the *explicit approach*, where composition rules are specified by humans; and the *implicit approach*, where training data is required to discover such rules [20]. In the literature, promising results using the *explicit approach* have employed *Hidden Markov Models* to generate computer music [21], but these are often limited to composing *counterpoint* or *harmonization* for previously existing tracks [22].

In this work, we present a musical application involving *SER's* minimum concurrency originally introduced in the conference version of this paper. Our technique consists of assembling pre-recorded *musical phrases* into a maximum-length loop of computer music, such that many rules fundamental to music theory are captured in the process. As a starting point, we focus on creating *blues* and *jazz* compositions, providing the finished result as an *open-source* simulation on the *Web browser*. In order to achieve this, however, we must first introduce the necessary terminology pertaining to music theory.

For the purpose of describing the following concepts, we have summarized their definitions from Schmidt-Jones's book [23]. First, a musical **phrase** corresponds to a group of individual notes that, together, express a definite melodic idea. It is customary for *phrases* to appear in pairs: the first *phrase* often sounds unfinished until it is completed by the second, almost as if the latter was answering a question posed by the former. *Phrases* that respect this dynamic are called **antecedent** and **consequent**, respectively.

A **bar** (or *measure*) is a group of *beats* that occur during a segment of time. When more than one independent melody takes place during the same *bar*, we call a piece of music **polyphonic** (e.g. Pachelbel's "Canon"; last chorus of "One Day More", from the musical "Les Miserables"). Finally, a **lick**, or *short motif*, corresponds to a brief musical idea that appears in many pieces of the same genre. In this work, a pair of *antecedent* and *consequent phrases*, when played sequentially, will also be referred to as a *lick*.

Having defined the necessary terminology, we now present the techniques employed to model the *resource graph*.

First, however, it is important to note that our focus is on assembling *musical phrases*, but this does not limit our strategy from being applied to other musical units such as chords or individual notes. In this context, we have chosen *musical phrases* from *jazz* and *blues* due to their "question" and "answer" nature, as we believe that they highlight the potential of this technique. Specifically, we would like to capture the following requirements:

(i) A **consequent** *phrase* may only be played after an **antecedent** *phrase*, forming a *lick*;

(ii) If two or more *phrases* are playing at the same time, either they are all **antecedent** or all **consequent**;

(iii) *Phrases* of different intensities (e.g. number of notes) may not go well together;

(iv) The final composition must be a loop, contain all available *phrases* and be of **maximum length**.

When scheduling *musical phrases*, we would like to be able to encode which *phrases* can be played sequentially and which can be played simultaneously. In a *resource graph*, an edge between any two nodes creates a neighborhood constraint, blocking these two nodes from simultaneous execution. As such, we represent *musical phrases* as nodes in the *resource graph* and connect them with an edge whenever they are unable to *operate* during the same *bar*, while still allowing them to be played sequentially.

Computationally, we structure each node as an object containing the following attributes: **Type** (e.g. *Antecedent*); **Genre** (e.g. *Blues*); **Note Count** (e.g. 8); and **File** (e.g. *antec02.mp3*). The *Note Count* attribute corresponds to the number of musical notes contained within a *phrase*, and will be used to approximately measure its intensity and satisfy requirement (iii). As such, for this specific example, we connect two nodes in the graph if and only if:

(1) they're of different types (*antecedent* and *consequent*);

(2) their *note count* is within a specified threshold;

(3) and they belong to the same genre.

As an additional feature, we would like to transition between *blues* and *jazz* in order to make the simulation more interesting. As such, we have also introduced **transitional phrases** which allow us to more seamlessly switch between these two genres. As a general rule, we connect these *transitions* with other *antecedent* phrases that lie within a given *note count* threshold, so that they themselves may act as a *consequent* phrase and provide closure to the current genre.

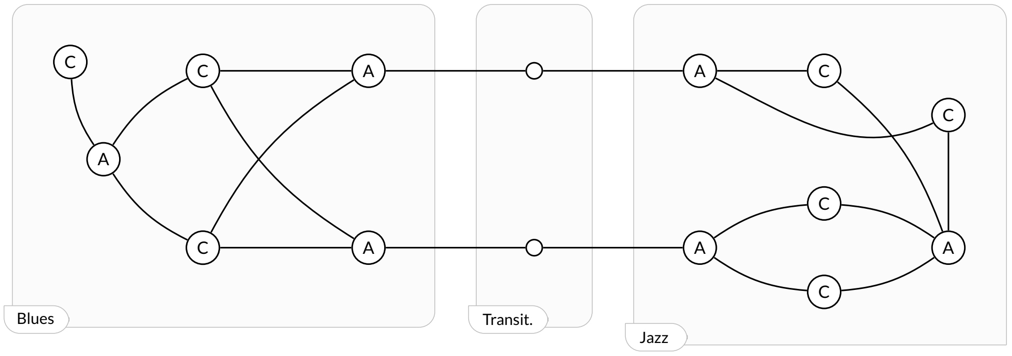


FIGURE 5 A *resource graph* where nodes marked as "A" and "C" represent *antecedent* and *consequent phrases*, respectively. Nodes connected by an edge are unable to be executed simultaneously, but can be played in sequence.

Figure 5 illustrates this entire model, where nodes marked as "A" and "C" are *antecedent* and *consequent*, respectively. The further a node is from a *transitional* node, the more *intense* is the *phrase* it represents. Because of the *antecedent / consequent* dynamic, the resulting graph is *bipartite*, for which the MINCON remains NP-complete [24]. In fact, when minimizing concurrency for this application, not only are we obtaining a maximum-length loop of computer music, but we are also controlling the level of *polyphony* within a song and preventing an oversaturation of sounds that may lead to excess noise throughout the composition. Additionally, it is important to note that initial orientations may violate requirement (ii), which states that *antecedent* and *consequent* nodes may not be played simultaneously. However, this is only an initialization issue, since the system will eventually reach a *period* and, as a consequence of the edge-reversal dynamics, the requirement will be satisfied: *antecedent phrases* will only become *sink* nodes when *consequent phrases* revert their edges, and vice versa.

Finally, we present a companion **simulation** playable *online* on any *Web* browser showcasing the edge-reversal dynamics when scheduling *musical phrases* (available at `https://cemarciano.github.io/Song-Generator/`), along with its open-source code (available at `https://github.com/cemarciano/Song-Generator`). The entire visualization was implemented in **JavaScript** using the following libraries: **Vis.js** [25], which allowed us to easily manipulate graphs and program the edge-reversal dynamics; and **Howler.js** [26], which handled the playback of sounds reliably for simultaneous files. All *musical phrases* have a length of 2 *bars* and were recorded on an electric guitar, while the *backing tracks* (the *rhythm sections*) were sampled from the sources specified in the simulation. Given that the *resource graph* in Figure 5 contains only 15 nodes, we manually synchronized each *phrase* with its corresponding starting time within a *bar* by using an offset. However, once synced, *phrases* may be played whenever a new 2-*bar* window starts. As a consequence, when setting the edge-reversal frequency to the duration of 2 *bars*, all *phrases* sound natural when played back.

5 | CONCLUSION

Concurrency is one of the central aspects when studying *Scheduling by Edge Reversal*, and directly impacts all of its distributed applications. In this work, for the first in the literature, we have reviewed most of the known real-world scenarios where the edge-reversal dynamics can be employed, providing an original example inspired by the *Shibuya Crossing* in *Tokyo*. The main contributions of this paper, however, were based on optimizing concurrency: for the maximization problem, we proved a negative result and proposed an approximation algorithm; for the minimization variant, we derived a much simpler expression for its direct calculation and provided a linear-time algorithm for obtaining an acyclic orientation that leads to minimum concurrency given a longest cycle as input, as well as computational results.

We also introduced a new application for *SER*, which revolves around assembling computer music, that directly benefits from minimum concurrency. Although the *musical phrases* employed in this work were recorded from a real electric guitar, the natural evolution of this technique would be to utilize *MIDI* files to build larger graphs. *MIDI* is a technical standard that allows a musical pattern to be described and synthesized by a computer [27], replacing the need for physical recording and manual synchronization. With this technique, sizeable *resource graphs* could be used to provide hour-long tracks of exclusively distinct music.

Lastly, in the future, we would like to achieve a feasible computational approach for calculating maximum concurrency to proven optimality. This could be done by either transforming this problem into another one (potentially related to the *multichromatic number* [28]), or developing an optimization model for maximum concurrency directly. Creating new applications that can benefit from the simplicity of *SER* is also an interesting theme for future research, especially when combined with new theoretical advancements for this technique.

ACKNOWLEDGEMENTS

We would like to thank the *European Network Optimization Group (ENOG)* for actively fomenting research on network optimization, creating a scientific environment that allowed for the existence of this paper. We would also like to thank the three anonymous reviewers from the *Scientific Committee* of the *9th International Conference on Network Optimization*, who provided constructive reviews and accepted the short version of this paper, titled "Minimum Concurrency for Assembling Computer Music", allowing us to publicly share this research with other members of the community. Finally, our most heartfelt thanks go to the jury of the *9th International Conference on Network Optimization*, who ultimately awarded the short version of this paper with the *Best Student Paper Award*, after a series of very encouraging compliments on the writing style of the paper as well as its presentation at the conference.

REFERENCES

[1] Barbosa VC, Gafni EM. Concurrency in Heavily Loaded Neighborhood-Constrained Systems. ACM Transactions on Programming Languages and Systems 1989 Oct;11(4):562–584. https://doi.org/10.1145/69558.69560.

[2] Arantes Jr GM. Trilhas, Otimização de Concorrência e Inicialização Probabilística em Sistemas sob Reversão de Arestas. PhD thesis, Federal University of Rio de Janeiro; 2006.

[3] Dijkstra EW. Hierarchical ordering of sequential processes. Acta Informatica 1971;1 (2):115–138. https://doi.org/10.1007/BF00289519.

[4] Tanenbaum AS. Modern Operating Systems. 3 ed. Upper Saddle River, NJ, USA: Pearson Prentice Hall; 2007.

[5] Carvalho D, Protti F, De Gregorio M, França FMG. A Novel Distributed Scheduling Algorithm for Resource Sharing Under Near-Heavy Load. Lecture Notes in Computer Science 2004;3544:431–442. https://doi.org/10.1007/11516798_31.

[6] Moscarini M, Petreschi R, Szwarcfiter JL. On node searching and starlike graphs. Congressus Numerantium 1998;131:75–84. DOI unavailable, must be requested directly from authors.

[7] Kirousis LM, Papadimitriou CH. Searching and pebbling. Lecture Notes in Computer Science 1986;47:205–218. https://doi.org/10.1016/0304-3975(86)90146-5.

[8] Gonçalves VCF, Lima PMV, Maculan N, França FMG. A Distributed Dynamics for WebGraph Decontamination. Lecture Notes in Computer Science 2010;6415:462–472. https://doi.org/10.1007/978-3-642-16558-0_39.

[9] Alves et al DSF. A Swarm Robotics Approach To Decontamination. In: Mobile Ad Hoc Robots and Wireless Robotic Systems: Design and Implementation Hershey, PA, USA: IGI Publishing; 2012.p. 107–122. https://www.doi.org/10.4018/978-1-4666-2658-4.ch006.

[10] Sotskov YN, Shakhlevich NV. NP-hardness of shop-scheduling problems with three jobs. Discrete Applied Mathematics 1995;59, no. 3:237–266. https://doi.org/10.1016/0166-218X(95)80004-N.

[11] Lengerke O, Acuña HG, Dutra MS, França FMG, Mora-Camino FAC. Distributed control of job-shop systems via edge reversal dynamics for automated guided vehicles. International Conference on Intelligent Systems and Applications 2012 Apr;1:25–30.

[12] Papadimitriou CH, Yannakakis M. Optimization, approximation, and complexity classes. J of Computer and System Sciences 1991 Dec;19 (3):425–440. https://doi.org/10.1016/0022-0000(91)90023-X.

[13] Bellare M, Goldreich O, Sudan M. Free bits, PCPS and non-approximability - towards tight results. SIAM J Comp 1998;27 (3):804–915. https://doi.org/10.1109/SFCS.1995.492573.

[14] Brooks RL. On coloring nodes of a network. Proc of Cambridge Philos Soc 1941;p. 194–197. https://doi.org/10.1017/S030500410002168X.

[15] Lovász L. A still better performance guarantee for approximate graph coloring. Information Processing Letters 1975 Dec;19 (3):269–271. https://doi.org/10.1016/0095-8956(75)90089-1.

[16] Marciano CE, Lucena A, Simonetti LG, França FMG. Minimum Concurrency for Assembling Computer Music. 9th International Conference on Network Optimization 2019;1:83–88. https://doi.org/10.5441/002/inoc.2019.16.

[17] Garey MR, Johnson DS. Computers and Intractability: A Guide to the Theory of NP-Completeness. New York, NY, USA, page 213: W. H. Freeman & Co.; 1979.

[18] Lucena A, Cunha A, Simonetti LG. A New Formulation and Computational Results for the Simple Cycle Problem. Electronic Notes in Discrete Mathematics 2013 Nov;44:83–88. http://doi.org/10.1016/j.endm.2013.10.013.

[19] Bollobás B. Random Graphs (Cambridge Studies in Advanced Mathematics). 2 ed. Cambridge University Press; 2001. https://doi.org/10.1017/CBO9780511814068.

[20] Shan MK, Chiu SC. Algorithmic compositions based on discovered musical patterns. Multimedia Tools and Applications 2010 Jan;46(1):1–23. https://doi.org/10.1007/s11042-009-0303-y.

[21] Nierhaus G. Algorithmic Composition: Paradigms of Automated Music Generation. Vienna, Austria: Springer-Verlag; 2009.

[22] Fernandez JD, Vico F. AI Methods in Algorithmic Composition: A Comprehensive Survey. Journal of Artificial Intelligence Research 2013 Nov;48(1):513–582. https://doi.org/10.1613/jair.3908.

[23] Schmidt-Jones C. Understanding Basic Music Theory. Houston, TX, USA: OpenStax CNX; 2007. http://cnx.org/content/col10363/1.3/.

[24] Krishnamoorthy MS. An NP-hard problem in bipartite graphs. SIGACT News 1975 Jan;7(1):26–26. https://doi.org/10.1145/990518.990521.

[25] Almende BV, et al, vis.js - A dynamic, browser based visualization library; 2015. http://visjs.org/, last accessed February 22, 2019.

[26] Simpson J, et al, howler.js - JavaScript audio library for the modern Web; 2013. https://howlerjs.com/, last accessed February 22, 2019.

[27] Huber DM. The MIDI Manual. 3rd ed. New York, NY, USA: Routledge; 2007. https://doi.org/10.4324/9780080479460.

[28] Stahl S. The multichromatic numbers of some Kneser graphs. Discrete Mathematics 1998 Apr;185(1):287–291. https://doi.org/10.1016/S0012-365X(97)00211-2.