Minimum Concurrency for Assembling Computer Music

Carlos E. Marciano
Federal University of Rio de Janeiro
Rio de Janeiro, RJ
cemarciano@poli.ufrj.br

Abilio Lucena
Federal University of Rio de Janeiro
Rio de Janeiro, RJ
abiliolucena@cos.ufrj.br

Felipe M. G. França
Federal University of Rio de Janeiro
Rio de Janeiro, RJ
felipe@ieee.org

Luidi G. Simonetti
Federal University of Rio de Janeiro
Rio de Janeiro, RJ
luidi@cos.ufrj.br

ABSTRACT

An effective algorithmic solution for resource-sharing problems in heavily loaded systems is *Scheduling by Edge Reversal (SER)*, essentially providing some level of concurrency by describing an order of *operation* for nodes in a graph. The resulting concurrency is a hard metric to optimize, as the decision problems associated with obtaining its *extrema* have been proved to be NP-complete. In this paper, we propose a novel approach involving longest cycles for solving the *Minimum Concurrency Problem* to proven optimality. Moreover, we show how this model can be used in the field of algorithmic composition to assemble a maximum-length loop of original computer music, capturing fundamental concepts in music theory. To illustrate this strategy, we present a complementary simulation accessible through the *Web*.

1 INTRODUCTION

Resource-sharing problems arise naturally in many scenarios, where graph algorithms are often employed to provide a distributed, asynchronous scheduling solution. By representing each process as a node, we define that nodes are connected by an edge if and only if they share a resource. Specifically, in neighborhood-constrained systems, a process is only allowed to *operate* if and only if all of its neighbors are *idle*, meaning that all of its required resources must be available at the time of *operation*. As a consequence, multiple processes requiring the same resource form a *clique*, a complete sub-graph in which only one node is allowed to *operate* at a time. A connected undirected graph representing resource dependencies among processes, as illustrated in Figure 1(a), will be referred to as a *resource graph* throughout this paper.

Under a heavy load assumption, where nodes are constantly demanding access to their required resources, an effective scheduling algorithm to ensure fairness and prevent starvation is *Scheduling by Edge Reversal (SER)*. Introduced by Gafni and Bertsekas [7] in 1981 and later formalized by Barbosa and Gafni [3] in 1989, *SER* has inspired many distributed resource-sharing applications ranging from asynchronous digital circuits [5] to the control of traffic lights in road junctions [4].

The execution of SER may be summarized as follows: by taking a directed acyclic graph *(DAG)* such as the one in Figure 1(b) as input, *SER* simultaneously *operates* all sinks, meaning that all nodes with no outgoing edges are allowed to utilize the resources they demand to perform their corresponding tasks. Once every sink is done *operating*, the orientation of their incoming edges is reverted, effectively allowing other nodes to become *sinks* themselves. This process is repeated indefinitely, as each new iteration will generate a new *DAG*, allowing different nodes to utilize resources and *operate*. Eventually, orientations will start repeating themselves, leading to the existence of periods. In fact, as observed by Barbosa and Gafni [3], all nodes operate the same number of times within a given period. Figure 1(c) illustrates this procedure.

In order to apply *SER* to any *resource graph* and obtain a corresponding schedule, an initial acyclic orientation must be generated. This initial *DAG* will directly impact the overall concurrency of the edge reversal procedure, leading to periods of different lengths and of different orientations. Intuitively, a highly concurrent dynamic will result in more nodes *operating* simultaneously while minimizing the amount of steps where each node is *idle*. Although a formal definition of concurrency is kept for Section 2, it's already inevitable to inquire about the complexity of problems such as obtaining the orientations that lead to the *extrema* of this metric. In fact, the decision problems associated with identifying the maximum as well as the minimum concurrency yielded by a given *resource graph* have been proved to be NP-complete by Barbosa and Gafni [3] and by Arantes Jr [11], respectively.

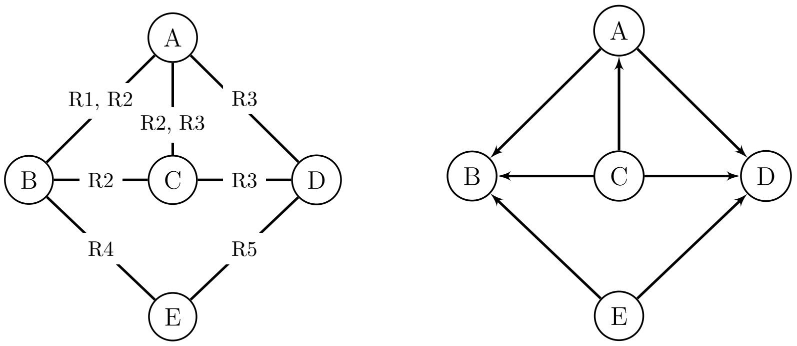
Contrary to intuition, obtaining the orientations of a resource graph from which *SER* will provide minimum concurrency is advantageous to a number of applications. For instance, Gonçalves et al. have employed *SER* under minimum concurrency to diminish the amount of *Web marshalls* needed for the distributed decontamination of *Webgraphs* [9, 14, 16], while Alves et al. have shown, through simulations of real conflagration scenarios, that less concurrency implies in a reduced number of automated firefighters required to control the flames [2]. However, despite *SER's* intrinsic connection to rhythms, no application in the field of algorithmic composition exists in the literature. As such, this paper presents a novel mechanism which, under minimum concurrency, schedules musical *phrases* to create the lengthiest possible original tracks that capture fundamental concepts in music theory, such as *rhythm* and *polyphony*. This is only possible by developing an optimization strategy for solving the *Minimum Concurrency Problem (MCP)*, which is also presented in this work as an original contribution.

The following is how the remainder of this paper is organized. In Section 2, we recall some graph-theoretic definitions associated with *SER*, including a formal metric for concurrency. Section 3, in turn, describes the concepts involved in our proposed reformulation of *MCP*. Finally, in Section 4, we show how minimum concurrency under *SER* can be used to assemble a maximum-length loop of computer music, expressing our concluding remarks and future work suggestions in Section 5.

Published in Proceedings of the International Network Optimization Conference (INOC), June 12-14, 2019:

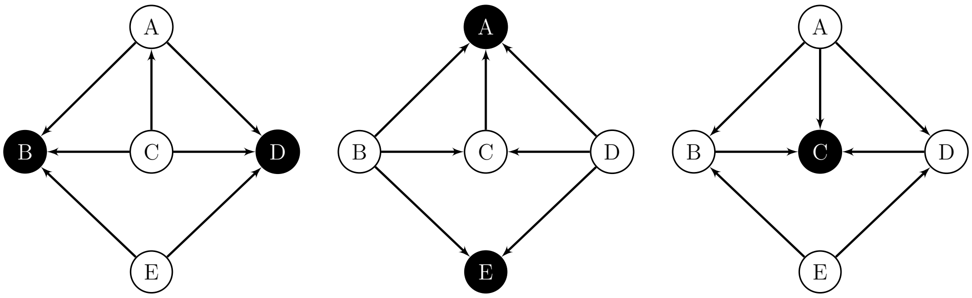
ISBN 978-3-89318-079-0 on OpenProceedings, ISSN 2510-7437

DOI: 10.5441/002/inoc.2019.16



(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.

2 GRAPH-THEORETIC BACKGROUND

Initially, as defined in Barbosa and Gafni [3], we shall characterize the necessary terminology to define *concurrency* under *SER.* As such, let $G = (V, E)$ be a connected undirected graph where $|E| \geq |V|$ (i.e. G is not a tree). Let $\kappa \subseteq V$ denote an undirected simple cycle in G, that is, a set of vertices that form a sequence of length $|\kappa| + 1$ of the form $i_0, i_1, ..., i_{|\kappa|-1}, i_0$. If κ is traversed from i_0 to $i_{|\kappa|-1}$, we say that it is traversed in the clockwise direction. Otherwise, we say that it is traversed in the counterclockwise direction. Let K denote the set of all simple cycles of G.

Moreover, an *acyclic orientation* of G is a function expressed as $\omega : E \longrightarrow V$ such that no undirected cycle κ of the form $i_0, i_1, ..., i_{|\kappa|-1}, i_0$ exists for which $\omega(i_0, i_1) = i_1$, $\omega(i_1, i_2) = i_2$, ..., $\omega(i_{|\kappa|-1}, i_0) = i_0$. Let Ω denote the set of all acyclic orientations of G.

Lastly, given an undirected simple cycle κ and an *acyclic orientation* ω, let $n_{cw}(\kappa, \omega)$ be defined as the number of edges oriented clockwise by ω in κ. Similarly, let $n_{ccw}(\kappa, \omega)$ be defined as the number of edges oriented by ω in the counterclockwise direction. Therefore, the *concurrency* of a graph G is defined as a function $\gamma : \Omega \longrightarrow \mathbb{R}$ such that:

$$\gamma(\omega) = \min_{\kappa \in K} \left\{ \frac{\min \{ n_{cw}(\kappa, \omega), n_{ccw}(\kappa, \omega) \}}{|\kappa|} \right\} \tag{1}$$

In other words, given an orientation ω, we check every simple undirected cycle κ of G and calculate the number of edges oriented in the clockwise direction as well as the number of edges oriented in the counterclockwise direction. We take the minimum of these two values and divide the result by the size of the undirected cycle κ. Whichever $\kappa \in K$ returns the smallest value will dictate the system's concurrency.

Finally, we must note that an equivalent result can also be obtained from a dynamic analysis. Let a *period* of length p be a sequence of distinct acyclic orientations $\alpha_0, ..., \alpha_{p-1}$ induced by the execution of *SER.* Let m be the number of times a node *operates* within a *period*, which is equal to all nodes. The expression $\gamma(\omega) = m/p$ is equivalent to Equation 1, despite being less significant to this paper. As an example, the concurrency provided by the schedule in Figure 1(c) is equal to 1/3, and can be obtained through both expressions.

3 OBTAINING MINIMUM CONCURRENCY

Our main goal in this section is to propose a linear-time algorithm for obtaining the minimum concurrency yielded by a resource graph G given one of its longest simple cycles as input. This reduction will essentially provide a computational model for the *Minimum Concurrency Problem (MCP),* allowing previously developed techniques for the *Longest Cycle Problem (LCP* [8]) to also be effective for *MCP.*

Initially, we shall derive a different expression for minimizing $\gamma(\omega)$ over all $\omega \in \Omega$. Given that Ω is a finite set, let γ^* denote the minimum value that Equation 1 assumes over all $\omega \in \Omega$:

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

The following lemma holds:

Lemma 3.1. $\gamma^* = \min_{\kappa\in K}\left\{\frac{1}{|\kappa|}\right\}$.

Proof. Consider Equation 1. For a given ω', let κ' be the simple cycle that minimizes the internal fraction. Let x be defined as $x = \min\{n_{cw}(\kappa',\omega'), n_{ccw}(\kappa',\omega')\}$, bringing Equation 1 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 2. 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 2 has become a minimization problem over all $\kappa \in K$. □

Lemma 3.1 is essentially the problem of finding a longest undirected cycle of G, whose minimum concurrency will be equal to the reciprocal of the size of its circumference.

We now show how to obtain ω^*, an orientation for which $\gamma(\omega^*) = \gamma^*$. Let κ^* be a longest simple cycle of G, meaning that $|\kappa^*| \geq |\kappa|$ for all $\kappa \in K$. The following theorem holds:

Theorem 3.2. *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 3.1 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 $1, ..., |\kappa^*|$ 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 $|\kappa^*| + 1, ..., |V|$ 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 $i_0, i_1, ..., i_0$ (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. □

Algorithm 1: 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 $\omega^*$
```

Finally, we structure the proof discussed in Theorem 3.2 as the algorithmic procedure presented in Algorithm 1. Its correctness relies on the aforementioned proof. Note that linear-time is attained only if the method *getClockwiseNeighborOf*(v) is $O(1)$. This will depend on the data structure used for storing κ^*, which is usually an array containing the vertices of the cycle in the order they should be visited. In this case, *getClockwiseNeighborOf*(v) will simply return the next element in the array and fulfill the $O(1)$ requirement. Since G is always a connected graph where $|E| \geq |V|$ as defined in Section 2, the overall time complexity of the algorithm is $O(m)$, where $m = |E|$.

4 ASSEMBLING COMPUTER MUSIC

As expressed by Shan and Chiu [19], effective computer music generation is the dream of computer music researchers. Previous explicit approaches (where composition rules are specified by humans) have resorted to *Hidden Markov Models* to capture the sequence requirements of melody [17], but are usually limited to composing *counterpoint* or *harmonization* for already existing tunes [6].

In this section, we show how a system under *SER's* minimum concurrency is capable of generating a maximum-length loop of pre-recorded musical *phrases*, while respecting fundamental concepts in music theory and creating original melodies for blues, jazz and rock music. In Subsection 4.1, we introduce the terminology that will be used throughout Subsection 4.2 to provide a strategy for representing musical *phrases* as graphs. Lastly, in Subsection 4.3, we discuss implementation-specific details for a complementary simulation included in Appendix A.

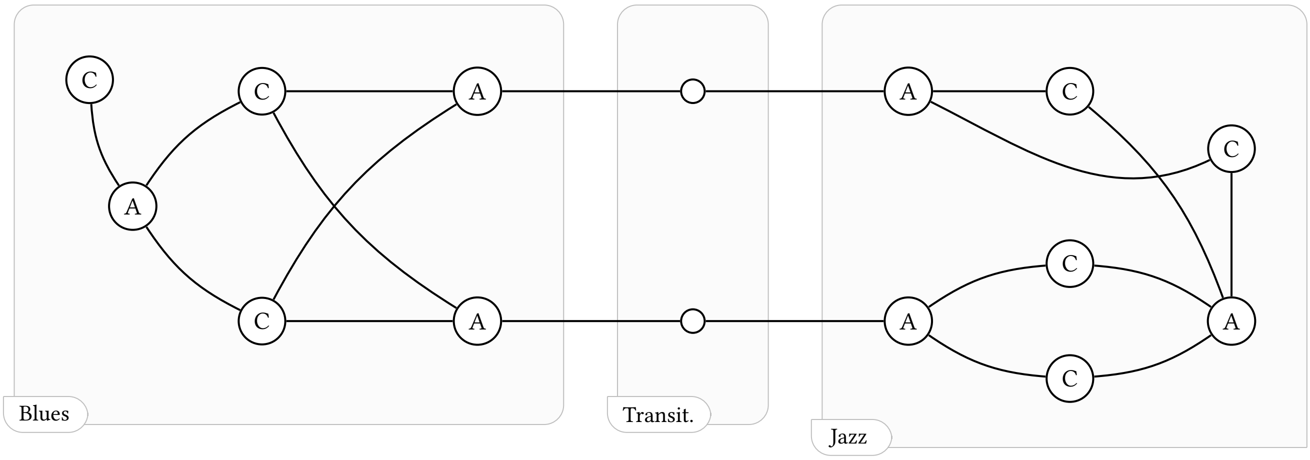


Figure 2: A *resource graph* where nodes marked as "A" and "C" represent *antecedent* and *consequent phrases*, respectively. Nodes connected by an edge are unable to be executed simultaneously, but are allowed to be played in sequence.

4.1 Music Theory Definitions

Initially, we shall define the necessary terminology from music theory employed throughout this section, for which we resort to Schmidt-Jones' book [18]. 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 were answering a question posed by the former. *Phrases* that respect this dynamic are called **antecedent** and **consequent**, respectively.

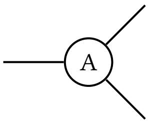
A **bar** (or *measure*) is a group of *beats* that occur during a segment of time. When more than one independent melody takes place during the same *bar*, we call a piece of music **polyphonic** (e.g. Pachelbel's "Canon"; last chorus of "One Day More", from the musical "Les Miserables"). Finally, a **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*.

4.2 Graph Representation

Although we believe that music generation through *SER* can be employed to assemble any musical unit (such as chords or individual notes) into a composition, the application we propose revolves around scheduling *phrases*. 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 arranging previously recorded (or generated) *phrases* into a graph, our goal is to structure which *phrases* can be played sequentially and which can be played simultaneously, creating a *polyphony*. By representing each *phrase* as a node, we are able to capture the aforementioned restrictions through the insertion of edges. In a *resource graph*, an edge between two nodes represents the inability of those nodes to *operate* at the same time. As such, an edge between two *phrases* is able to prevent them from occurring during the same *bar*, while allowing each separate *phrase* to be played in sequence.



(a) An *antecedent* phrase.

Property	Value
Type	*Antecedent*
Genre	Blues
Note Count	8
File	antec02.mp3

(b) Node attributes.

Figure 3: An example of a node and its attributes.

Above, in Figure 3, we present the information contained within each node. A *note count*, corresponding to the number of notes within a phrase, is used to measure its intensity. For this specific example, two nodes will be connected to each other 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.

Moreover, we'd like to make this example more interesting by allowing a transition between two different genres: *blues* and *jazz*. By introducing **transitional phrases** that incorporate elements from both genres, a more seamless changeover can be achieved. *Antecedent phrases* from *blues* and *jazz*, when connected to *transitional nodes*, can act as gateways that allow access to their respective genres.

Figure 2 illustrates all the previously discussed components. Nodes marked as "A" and "C" represent *antecedent* and *consequent phrases*, respectively. The further a node is from a *transitional* node, the more *intense* is the *phrase* it represents. Due to the *antecedent / consequent* dynamic, the resulting graph is *bipartite*, for which *MCP* remains NP-complete [12].

4.3 Implementation Details

In order to demonstrate the ideas discussed in Subsection 4.2, we have developed a simulation showing how the *phrase*-scheduling dynamic, when applied to a graph such as the one in Figure 2, is able to produce musical loops of maximum length. In this subsection, we document our steps and discuss implementation details that may be useful for future work. The final result, featuring the *resource graph* from Figure 2, is presented in Appendix A.

As discussed in Section 3, the first step in the process of obtaining minimum concurrency is identifying a longest simple cycle. Although this task is visually straightforward when considering the *resource graph* from Figure 2, larger instances require a computational approach. As such, we relied on the *Simple Cycle Problem* branch-and-cut strategy proposed in Lucena, Cunha and Simonetti [15], which is based on a formulation that decomposes simple cycles into one simple path and an additional edge. We implemented this procedure in the **C** programming language and used the *XPRESS Mixed Integer Programming* package to solve linear programs and manage the branch-and-cut tree.

Despite the example from Figure 2 only containing 15 nodes, our computational results have shown that the aforementioned strategy is able to solve, in under 1 hour, instances of random graphs with as many as 2 000 nodes and 40 034 edges (probability $p = 0.01$ for an edge to exist between two nodes), being an appealing approach for larger instances. In turn, a linear-time implementation of Algorithm 1 is employed to provide an acyclic orientation for the *resource graph*, yielding minimum concurrency. The pipeline presented in Figure 4 summarizes this process.

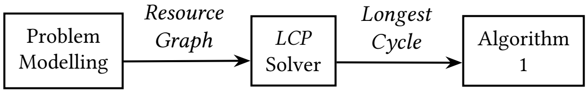


Figure 4: Implementation pipeline for solving *MCP*.

Note, however, that initial orientations may violate requirement (ii), which states that *antecedent* and *consequent* phrases are not allowed to be played together. This is because sinks may be formed anywhere in the graph when orienting nodes outside the original longest cycle. However, this is merely an initialization issue: once a *SER period* is reached, the system will enforce, through the edge-reversal dynamic, that *antecedent phrases* will only become sink nodes when a *consequent phrase* reverts its edges, and vice versa.

Having attained minimum concurrency for the *resource graph* in Figure 2, we switched our attention to developing a visualization strategy. From a compatibility perspective, a web simulation built in **JavaScript** is both lightweight and easy to access on most platforms. Moreover, two convenient libraries, available under the *MIT License*, made this choice even more appealing: **Vis.js** [1], which enabled us to visually represent any graph and handle the necessary edge-reversal dynamics; and **Howler.js** [20], providing a reliable audio interface when dealing with multiple files.

Finally, we curated audio recordings responsible for the *rhythm* sections (also known as *backing tracks*) and recorded all *antecedent* and *consequent phrases* on an electric guitar. Given that this small simulation is comprised of only 15 nodes, the process of syncing each *phrase* to their corresponding *backing track* was performed manually. For instance, a *12-Bar Blues* composition may alternate between *antecedent* and *consequent phrases* every 2 *bars*. Different *phrases* have different starting points within this window, requiring an offset to account for synchronization. However, once synced, *phrases* may be played whenever a new *2-bar* window starts. As such, by setting the edge-reversal frequency to *2 bars*, every *phrase* will sound natural when their corresponding node becomes a sink.

5 CONCLUSION

In this paper, two main contributions to *SER* were presented: first, we reformulated the *Minimum Concurrency Problem*, providing a viable approach for its optimization and allowing many empirically attractive *LCP* solvers to also be effective for *MCP*. Secondly, we proposed a novel strategy for assembling original computer music, which schedules all available *building blocks* (in our example, musical *phrases*) into a maximum-length loop, all the while incorporating essential music-theoretical restrictions.

Regarding *SER's* debut in algorithmic composition, we are eager to discover how other researchers and musicians may employ this technique and its variations to create unique songs. We note that the *Web* is a never-ending repository of musical *phrases*, many of which are encoded in *MIDI* format. *MIDI* is a technical standard that allows a musical pattern to be described and synthesized by a computer [10], replacing the need for physical recording and manual synchronization. This gain in development speed can allow for the modelling of truly large *resource graphs*, producing hour-long tracks of exclusively distinctive music.

Another aspect that can be investigated is controlling the level of *polyphony* within a song. For instance, higher concurrency values imply in a large number of independent melodies occurring during the same *bar*, which may lead to undesirable noise throughout the composition. As such, minimum concurrency not only provides a maximum-length loop of music, but also avoids an oversaturation of sounds that may lead to low-quality *polyphony*. Currently, we investigate how **octave** information (the frequency range in which the fundamental pitch of each note is found) can be used to control which sounds should be played simultaneously (*e.g.:* a phrase whose notes were recorded near *octave* C_3 could be played alongside a phrase with notes situated around *octave* C_5). This approach would avoid melody lines competing for the same frequency range, leading to more distinguishable and pleasant sounds.

Lastly, we invite other researchers to investigate a viable computational model for the *Maximum Concurrency Problem*, which consists of maximizing Equation 1 over the set of acyclic orientations Ω. This breakthrough would impact many distributed resource-sharing applications, such as routing *Automated Guided Vehicles (AGVs)* [13], scheduling job shop tasks [13] and controlling traffic lights in road junctions [4]. Naturally, new engaging applications that could benefit from *SER's* simplicity are also an interesting theme for future research, especially when combined with new theoretical advancements for this technique.

A MUSICAL SIMULATION

The musical simulation referred to throughout this paper is available online at the following website, and can be viewed in any browser: https://cemarciano.github.io/Song-Generator/.

This simulation is an open-source project distributed under the *GNU GPL v3.0 License*. Source code is available at the following website: https://github.com/cemarciano/Song-Generator.

REFERENCES

[1] B. V. Almende et al. 2015. vis.js - A dynamic, browser based visualization library. (2015). Retrieved February 22, 2019 from http://visjs.org/

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

[3] Valmir C. Barbosa and Eli M. Gafni. 1989. Concurrency in Heavily Loaded Neighborhood-Constrained Systems. *ACM Transactions on Programming Languages and Systems* 11, 4 (Oct. 1989), 562–584. https://doi.org/10.1145/69558.69560

[4] D. Carvalho, Fábio Protti, Massimo De Gregorio, and Felipe M. G. França. 2004. A Novel Distributed Scheduling Algorithm for Resource Sharing Under Near-Heavy Load. *Lecture Notes in Computer Science* 3544 (2004), 431–442. https://doi.org/10.1007/11516798_31

[5] Ricardo F. Cassia, Vladmir C. Alves, Frederico G. Besnard, and Felipe M. G. França. 2009. Synchronous-To-Asynchronous Conversion of Cryptographic Circuits. *Journal of Circuits, Systems and Computers* 18, 2 (2009), 271–282. https://doi.org/10.1142/S0218126609005058

[6] Jose D. Fernandez and Francisco Vico. 2013. AI Methods in Algorithmic Composition: A Comprehensive Survey. *Journal of Artificial Intelligence Research* 48, 1 (Nov. 2013), 513–582. https://doi.org/10.1613/jair.3908

[7] Eli M. Gafni and Dimitri P. Bertsekas. 1981. Distributed Algorithms for Generating Loop-Free Routes in Networks with Frequently Changing Topology. *IEEE Transactions on Communications* 29, 1 (Jan. 1981), 11–18. https://doi.org/10.1109/TCOM.1981.1094876

[8] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, page 213.

[9] Vanessa C. F. Gonçalves, Priscila M. V. Lima, Nelson Maculan, and Felipe M. G. França. 2010. A Distributed Dynamics for WebGraph Decontamination. *Lecture Notes in Computer Science* 6415 (2010), 462–472. https://doi.org/10.1007/978-3-642-16558-0_39

[10] David Miles Huber. 2007. *The MIDI Manual* (3rd ed.). Routledge, New York, NY, USA. https://doi.org/10.4324/9780080479460

[11] Gladstone M. Arantes Jr. 2006. *Trilhas, Otimização de Concorrência e Inicialização Probabilística em Sistemas sob Reversão de Arestas*. Ph.D. Dissertation. Federal University of Rio de Janeiro, Rio de Janeiro, Brazil. https://www.cos.ufrj.br/index.php/pt-BR/publicacoes-pesquisa/details/15/2039

[12] M. S. Krishnamoorthy. 1975. An NP-hard problem in bipartite graphs. *SIGACT News* 7, 1 (Jan. 1975), 26–26. https://doi.org/10.1145/990518.990521

[13] Omar Lengerke, HernÃąn G. AcuÃśa, Max S. Dutra, F. M. G. FranÃğa, and Felix A. C. Mora-Camino. 2012. Distributed control of job-shop systems via edge reversal dynamics for automated guided vehicles. *International Conference on Intelligent Systems and Applications* 1 (April 2012), 25–30. https://doi.org/10.13140/RG.2.1.3054.5127

[14] Linda Luccio, Fabrizio annd Pagli. 2007. Web Marshals Fighting Curly Link Farms. *Lecture Notes in Computer Science* 4475 (2007), 240–248. https://doi.org/10.1007/978-3-540-72914-3_21

[15] Abilio Lucena, Alexandre Cunha, and Luidi G. Simonetti. 2013. A New Formulation and Computational Results for the Simple Cycle Problem. *Electronic Notes in Discrete Mathematics* 44 (Nov. 2013), 83–88. https://doi.org/10.1016/j.endm.2013.10.013

[16] Marina Moscarini, Rossella Petreschi, and Jayme L. Szwarcfiter. 1998. On node searching and starlike graphs. *Congressus Numerantium* 131 (1998), 75–84. DOI unavailable, must be requested directly from authors.

[17] Gerhard Nierhaus. 2009. *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer-Verlag, Vienna, Austria. https://doi.org/10.1007/978-3-211-75540-2

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

[19] Man-Kwan Shan and Shih-Chuan Chiu. 2010. Algorithmic compositions based on discovered musical patterns. *Multimedia Tools and Applications* 46, 1 (Jan. 2010), 1–23. https://doi.org/10.1007/s11042-009-0303-y

[20] James Simpson et al. 2013. howler.js - JavaScript audio library for the modern Web. (2013). Retrieved February 22, 2019 from https://howlerjs.com/