Minimum Concurrency via Maximal Cycles

Carlos E. Marciano [1,2] Abilio Lucena [3] Felipe M. G. França [4] Luidi G. Simonetti [5]

Programa de Engenharia de Sistemas e Computação
Universidade Federal do Rio de Janeiro
Rio de janeiro, Brasil

Abstract

An effective algorithmic solution for resource-sharing problems in heavily loaded systems is Scheduling by Edge Reversal, 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, since the decision problems associated with obtaining its *extrema* have been proved to be NP-complete. In this paper, we propose a novel approach involving maximal cycles for attaining minimum concurrency, which is desired for problems associated with graph decontamination. We prove the correctness of our algorithm and implement our techniques, showing that reasonably large instances may be solved to proven optimality under acceptable CPU times.

Keywords: Minimum concurrency, Maximal cycles, Scheduling by edge reversal

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 [1] in 1981 and later formalized by Barbosa and Gafni [2] in 1989, *SER* has inspired many distributed resource-sharing applications ranging from asynchronous digital circuits [3] to the control of traffic lights present 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 their corresponding resources to perform a given task. 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

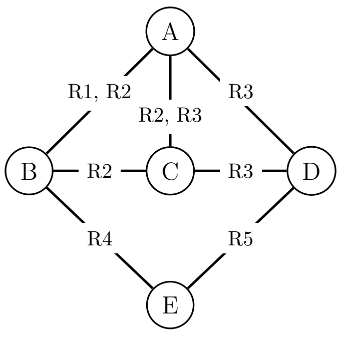
[1] The authors would like to thank CNPq for partially funding this research.

[2] Email: cemarciano@poli.ufrj.br

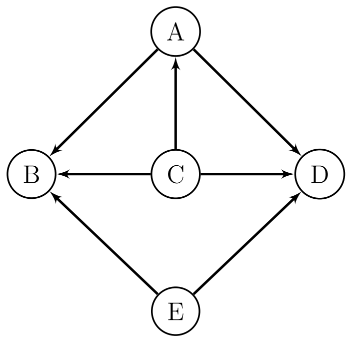
[3] Email: abiliolucena@cos.ufrj.br

[4] Email: felipe@cos.ufrj.br

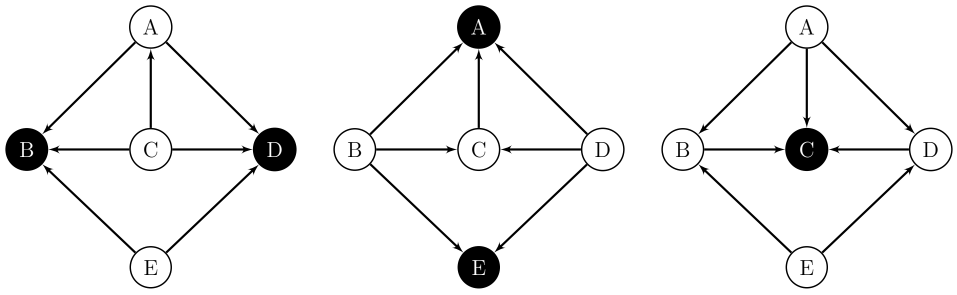
[5] Email: luidi@cos.ufrj.br



(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.

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

to the existence of periods. In fact, as has been observed by Barbosa and Gafni, 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 commencing *DAG* will directly impact the overall concurrency obtained through the edge reversal procedure, leading to periods of different lengths and of different orientations. Intuitively, a highly concurrent dynamic will present more nodes *operating* simultaneously while minimizing the amount of steps where each node is *idle*. Although concurrency will be better defined in section 2, it's already inevitable to inquire the complexity of problems such as obtaining the orientations that lead to the *extrema* of this metric. In fact, the decision problems associated with obtaining 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 [2] and by Arantes Jr [5], 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, in the decontamination of circulant graphs representing link farms in the Web, where links to a target web page are spread throughout other pages to promote visibility [6], one goal is to minimize the number of *web marshals* performing the decontamination procedure. Moscarini, Petreschi and Szwarcfiter [7] have also studied a generalization of this process for starlike graphs. Under a dynamic in which a node becomes healthy once it is no longer exposed to contaminated neighbors, it is shown in Gonçalves et al. [8] that lower concurrency values imply in a reduced number of *web marshals* employed in the dismantling effort. This decontamination dynamic can also be extended to conflagration scenarios, where firefighters must guard a safe node until its neighbors are cleared of fire. In Alves et al. [9], the use of robots to extinguish the flames within an apartment was simulated through the dynamics of *SER* decontamination.

The following is how the remainder of this paper is organized. In section 2, we recall some graph-theoretic definitions associated with Scheduling by Edge Reversal, including a metric for concurrency. Section 3 is our main theoretical contribution, showing that minimum concurrency can be obtained in linear-time given a previously discovered maximal cycle. Finally, in section 4, we implement a two-phase algorithm for finding minimum concurrency of various relevant instances, expressing our concluding remarks in section 5.

2 Graph-Theoretic Background

Initially, as has been defined in Barbosa and Gafni [2], 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 κ denote an undirected simple cycle in G, that is, a sequence of $|\kappa|$ vertices $i_0, i_1, ..., i_{|\kappa|-1}, i_0$. 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 of the form $\omega : E \rightarrow V$ such that no undirected cycle $i_0, i_1, ..., i_{k-1}, i_0$ exists for which $\omega(i_0, i_1) = i_1$, $\omega(i_1, i_2) = i_2$, ..., $\omega(i_{k-1}, i_0) = i_0$. 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 $\gamma : \Omega \rightarrow \mathbb{R}$ such that:

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

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 finding the minimum concurrency yielded by a resource graph G given one of its maximal simple cycles as input, as well as analyzing the correctness of such algorithm. Initially, we shall derive a different expression for minimizing $\gamma(\omega)$ for all $\omega \in \Omega$, showing how the problem of finding an orientation that yields minimum concurrency is closely related to the problem of obtaining an undirected maximal cycle of G.

Equation 1 is a function of ω. Given that Ω is a finite set, let γ^* denote the minimum value that Equation 1 assumes for every $\omega \in \Omega$. We may then write an expression for minimum concurrency as follows:

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

However, for every cycle $\kappa \in K$, there will always exist an acyclic orientation $\omega \in \Omega$ such that $n_{cw}(\kappa, \omega) = 1$ and $n_{ccw}(\kappa, \omega) = |\kappa| - 1$. This follows immediately from the fact that a directed cycle would only exist if and only if either n_{cw} or n_{ccw} were equal to zero for a given κ. As such, since the numerator in Equation 2 may assume the value of 1 for every $\kappa \in K$, it is no longer a function of ω and we may rewrite our expression for γ^* as follows:

$$\gamma^* = \underset{\kappa \in K}{min} \left\{ \frac{1}{|\kappa|} \right\} \tag{3}$$

Equation 3 is essentially the problem of finding a maximal undirected cycle of G, whose minimum concurrency will be equal to the reciprocal of the size of its circumference. However, although we have shown how to calculate γ^*, it still unclear how to obtain ω^*, an orientation for which $\gamma(\omega^*) = \gamma^*$. As such, the remainder of this section is dedicated to describing an algorithm for finding ω^* given a maximal cycle of G, as well as a brief discussion of its correctness and complexity.

Let κ^* be a maximal simple cycle of G, meaning that $|\kappa^*| \geq |\kappa|$ for all $\kappa \in K$. The following theorem holds:

Theorem 3.1 *Given any maximal 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 for all $\omega \in \Omega$.*

Algorithm 1: A linear-time algorithm for finding an acyclic orientation that leads to minimum concurrency given a maximal cycle as input.

```
Input  : An undirected graph $G = (V, E)$ and its maximal cycle $\kappa^*$
Output: An 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 $\omega^*(u, v) = v$
    end
    else
        Orient edge such that $\omega^*(u, v) = u$
    end
end
return $\omega^*$
```

.

Proof. Equation 3 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 $n_{ccw}(\kappa^*, \omega^*) = |\kappa^*| - 1$, where κ^* is a maximal cycle. Orienting κ^* under the aforementioned conditions can be performed in linear-time by traversing the cycle $\kappa*$ 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 higher identification numbers, only one edge (connecting the vertices with the highest and the lowest identification numbers) will be oriented in the opposing 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 higher 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 higher identification numbers, it is impossible to return to i_0 after leaving it, for every $i_0 \in V$. As such, no cycles are formed. □

Finally, we structure the proof discussed in Theorem 3.1 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 $\kappa*$, 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 $|E| > |V|$ as defined in section 2, the overall time complexity of the algorithm is $O(m)$, where $m = |E|$.

4 Experimental Results

In this section, we implement a two-phase algorithm for obtaining acyclic orientations that attain minimum concurrency. Since the decision problem associated with finding a maximal simple cycle of G is NP-complete [10], the first phase of our algorithm consists of employing a branch-and-cut optimization strategy for identifying one of these cycles. The second phase, in turn, collects this result and applies Algorithm 1 to obtain an acyclic orientation that results in minimum concurrency. Out of our computational experiments, we show that reasonably large instances related to the graph decontamination problem discussed in section 1 may be solved to proven optimality under acceptable CPU times.

For the first phase of our algorithm, we rely on the Simple Cycle Problem branch-and-cut strategy proposed in Lucena, Cunha and Simonetti [11], which is based on a formulation that decomposes simple cycles into one simple path and an additional edge, connecting the extreme vertices of the path. This algorithm separates Generalized Subtour Elimination Constraints *on the fly* (as they become violated at a linear programming relaxation in the branch-and-cut enumeration tree).

All implementation was done in the C programming language and uses the *XPRESS Mixed Integer Programming* package v8.5.3 to solve linear programs and manage the branch-and-cut tree. All other features of the solver, such as pre-processing, primal heuristics and automatic cut generation were kept switched-off. Experiments were conducted on an Intel Core i9-8950HK based machine with 16 Gbytes of RAM memory, running Linux Ubuntu 18.04.1, with only one thread being used.

In order to assess the viability of our proposal, two sets of instances were selected. For the first set, given $n = |V|$, we created circulant graphs of the form $Ci_n(1, ..., \lfloor n/2 \rfloor)$, where each edge was generated with a given probability p (*e.g.* for $p = 1$, we have the complete graph K_n. For $p = 0$, no edges exist). In case multiple components were formed, we would randomly connect them at the end of the process so as to ensure that our program would always generate connected graphs. This probabilistic nature of our instances allowed us to control their density and capture less predictable structures.

For the second set of instances, so as to evaluate the feasibility of our strategy for graphs that are known to be more challenging, we selected hamiltonian circuit instances present in Heidelberg University's Discrete and Combinatorial Optimization group website [12]. However, due to limiting our program to a total of 3600 seconds of execution, we could only solve 2 out of the 9 available instances.

Given that our intent is to simply demonstrate feasibility, all instances were executed once, with a time limit set to 1 hour of execution and CPU Time (in seconds) rounded to the closest integer. The results for the first and second set of instances are displayed in Table 1 and Table 2, respectively.

Nodes	**Edges**	p	$\lvert\kappa^*\rvert$	**Min Concurrency**	**CPU Time (s)**
200	392	0.01	183	1/183	1
200	3826	0.1	200	1/200	2
1000	1912	0.002	882	1/882	169
1000	19912	0.02	1000	1/1000	552
1000	180151	0.2	-	-	> 3600
2000	4079	0.001	1807	1/1807	874
2000	40034	0.01	2000	1/2000	3599
2000	380147	0.1	-	-	> 3600
2000	1999000	1	-	-	> 3600

Table 1
Minimum concurrency for generated circulant graphs.

Instance Name	**Nodes**	**Edges**	**Min Concurrency**	**CPU Time (s)**
alb1000	1000	1998	1/1000	23
alb3000e	3000	5996	1/3000	2842

Table 2
Minimum concurrency for challenging hamiltonian circuit instances.

Our results show that, despite sizable instances with higher density values still being a challenge, we were able to solve considerably large graphs in under 1 hour. Given that many graph decontamination instances arise in much more compact and sparse environments (such as the the firefighting example from Alves et al. [9]), we conclude that our algorithm was proven to be a viable approach for obtaining minimum concurrency in practical scenarios.

5 Conclusion

We have presented a novel technique for obtaining minimum concurrency under Scheduling by Edge Reversal, being particularly useful in scenarios derived from graph decontamination problems. By first identifying a maximal cycle of the *resource graph* G, we have demonstrated how to obtain, in linear-time, an orientation that attains minimum concurrency. Despite the decision problem associated with finding maximal cycles still being NP-complete, we have shown through experimental results that modern branch-and-cut techniques are able to solve significantly large instances under acceptable CPU times.

References

[1] Gafni, E. M., and D. P. Bertsekas, *Distributed Algorithms for Generating Loop-Free Routes in Networks with Frequently Changing Topology*, IEEE Transactions on Communications **29(1)** (1981), 11–18.

[2] Barbosa, V. C., and E. M. Gafni, *Concurrency in Heavily Loaded Neighborhood-Constrained Systems*, ACM Transactions on Programming Languages and Systems **11(4)** (1989), 562–584.

[3] Cassia, R. F., and V. Alves, and F. Besnard, and F. M. G. França, *Synchronous-To-Asynchronous Conversion of Cryptographic Circuits*, Journal of Circuits, Systems and Computers **18(2)** (2009), 271–282.

[4] Carvalho, D. and F. Protti and M. De Gregorio and F. M. G. França, *A Novel Distributed Scheduling Algorithm for Resource Sharing Under Near-Heavy Load*, Lecture Notes in Computer Science **3544** (2004), 431–442.

[5] Arantes Jr, G. M., "Trilhas, Otimização de Concorrência e Inicialização Probabilística em Sistemas sob Reversão de Arestas", Ph.D. thesis, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2006.

[6] Luccio, F. and L. Pagli, *Web Marshals Fighting Curly Link Farms*, Lecture Notes in Computer Science **4475** (2007), 240–248.

[7] Moscarini, M., and R. Petreschi, and J. L. Szwarcfiter, *On node searching and starlike graphs*, Congressus Numerantium **131** (1998), 75–84.

[8] Gonçalves, V. C. F. and P. Lima, and N. Maculan, and F. M. G. França, *A Distributed Dynamics for WebGraph Decontamination*, Lecture Notes in Computer Science **6415** (2010), 462–472.

[9] Alves, D. S. F., et al., "A Swarm Robotics Approach To Decontamination", Mobile Ad Hoc Robots and Wireless Robotic Systems: Design and Implementation, 1st ed., IGI Publishing Hershey, PA, USA (2012), 107–122.

[10] Garey, M. R., and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", W. H. Freeman & Co. New York, NY, USA (1979), p213.

[11] Lucena, A., and A. Cunha, and L. Simonetti, *A New Formulation and Computational Results for the Simple Cycle Problem*, Electronic Notes in Discrete Mathematics, **44** (2013), 83–88.

[12] Discrete and Combinatorial Optimization group, Heidelberg University, "Hamiltonian cycle problem data", accessed 11/25/2018. URL: https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/hcp/