DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Approximation Algorithms for Loop-Free Updates in Software-Defined Networks

Radu Vintan

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Approximation Algorithms for Loop-Free Updates in Software-Defined Networks

Approximationsalgorithmen für Routenaktualisierungen in Software-Defined Networks

Author:	Radu Vintan
Supervisor:	Prof. Harald Räcke
Submission Date:	15.08.2022

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Garching, 15.08.2022 Radu Vintan

Acknowledgments

I am very grateful to my supervisor, Professor Harald Räcke, for advising both my Guided Research project and my Master's Thesis. He had the generosity to meet me regularly during the last year and we often tackled hard problems together even if this implied hours of frustration. I feel that I have learned a lot from him about the mental attitude which is required to be a successful problem solver while also having fun. I hope to be able to follow his example successfully in my future research.

Professor Stefan Schmid from TU Berlin proposed useful directions to investigate and offered valuable advice when it was not clear what to do next. I am thankful to him for his openness and friendly feedback.

Abstract

We study two variants of the more general Software-defined Network (SDN) update problem. Two different paths from a source s to a destination d are given, consisting of the same intermediary routers listed in two different orders. The forwarding tables of all routers need to be updated in order to switch from a routing along the old path to a routing along the new path. During the update no cycles are allowed to exist at any time for the Strong Loop-Freedom (SLF) Problem, while for the Relaxed Loop-Freedom (RLF) Problem only cycles reachable from the source s are forbidden. The objective is to update all routers as fast as possible while ensuring the respective loop-freedom property. We design approximation algorithms for both problems, analyze their performance in practice and try to prove approximation guarantees. One of our techniques can also be used to compute the optimal solution for instances of significant size faster than with any other method we know. We present both our theoretical ideas and the results of our publicly available implementation. In particular, it appears empirically that SLF instances are often well approximated by an appropriate Linear Programming rounding procedure, while RLF is best approximated by a purely combinatorial Local Search algorithm.

Kurzfassung

Wir untersuchen zwei Varianten des allgemeineren Software-defined Network (SDN) Update Problems. Es sind zwei verschiedene Pfade von einer Quelle s zu einem Ziel d gegeben, die aus denselben Zwischenroutern bestehen, und welche in zwei verschiedenen Reihenfolgen aufgeführt sind. Die Forwarding-Tabellen aller Router müssen aktualisiert werden, um von einem Routing entlang des alten Pfades zu einem Routing entlang des neuen Pfades zu wechseln. Während der Aktualisierung dürfen, für das Strong Loop-Freedom (SLF) Problem, zu keinem Zeitpunkt Zyklen existieren, während für das Relaxed Loop-Freedom (RLF) Problem nur Zyklen verboten sind, die von der Quelle s erreichbar sind. Das Ziel ist alle Router so schnell wie möglich zu aktualisieren und dabei die jeweilige zykelfrei-Eigenscheft zu gewährleisten. Wir haben Approximationsalgorithmen für beide Aufgaben entwickelt, ihre Leistung in der Praxis analysiert und versucht, Approximationsgarantien zu beweisen. Mit einem unserer Verfahren lässt sich auch die optimale Lösung für Instanzen von signifikanter Größe schneller berechnen als mit jeder anderen von uns bekannten Methode. Wir erklären sowohl unsere theoretischen Ideen als auch die Ergebnisse unserer öffentlich zugänglichen Implementierung. Insbesondere zeigt sich empirisch, dass SLF-Instanzen oft gut durch ein geeignetes Rundungsverfahren der Linearen Programmierung approximiert werden, während RLF am besten durch einen rein kombinatorischen Local Search Algorithmus approximiert wird.

Contents

1 Introduction

A new perspective in computer networking has appeared with the advent of Software-defined Networks (SDN), where the control over a set of routers/switches is delegated to a central authority. The objective of this controller is to change the routing in the network, while at the same time ensuring that certain consistency guarantees are met. However, in many cases, fulfilling consistency comes at the price of speed and flexibility, which explains why a rich literature developed around the problems raised by this inherent need of compromise. The reference [1] contains a survey both of problems and of (partial) solutions that have been developed over the years in this area.

Let us consider more concretely a challenge posed by any SDN implementation. When the controller updates a route r_1 to a new route r_2, it needs to update the forwarding tables of all involved routers/switches. If it tries to update all nodes at once, some nodes might update faster than others, resulting in dangerous temporary scenarios where some nodes use the old rules and other nodes use the new rules. Such hybrid routes usually violate the required consistency properties. Hence, the controller needs to find a suitable strategy to avoid these issues.

One possible strategy, designed by Reitblatt et al. [2], uses a *2-Phase Commit Protocol (2PC)* with *packet tagging*. In the first phase, all routers are informed of the new forwarding rules they should use. However, the routers do not switch to the new rules immediately. Instead, in the second phase, the controller begins to mark the packets which should use the new route. Then, if a router receives an unmarked packet it uses the old rule, and if it receives a marked packet it uses the new rule. This elegant approach guarantees that any packet is routed either entirely on the old route or entirely on the new route, and hence no hybrid routes are used. Still, practice has shown that tagging is undesirable and can lead to unexpected issues as well [3].

Another strategy is to group updates cleverly, i.e. to find *transiently consistent (SDN) updates*. More precisely, the controller partitions the set V of routers into k subsets V_1 to V_k. Then, in

any *round i*, the controller sends updates to the routers in V_i and then waits long enough to guarantee that all these routers have updated, before beginning with *round* $i+1$. The sets V_1 to V_k are chosen in such a way that, assuming V_1 to V_i have already updated, any order in which the routers in V_{i+1} update guarantees consistency at all times. In the following we review the literature around this technique.

There are at least two optimization problems one can formulate for finding *transiently consistent updates*. One such problem tries to maximize the number $|V_i|$ of routers which are updated concurrently in some round i (which we can w.l.o.g. assume to be the round $i = 1$). In [4], Wattenhofer et al. proposed an algorithm for this problem where they guarantee *loop-freedom*. Loop-freedom is a consistency property which guarantees that no packet ends up in a loop at any time. Later, Schmid et al. [5] made a distinction between *strong loop-freedom (SLF)* and *relaxed loop-freedom (RLF)*. In the former, no cycles are allowed at all in the network (even if they are unreachable by the traveling packets), while in the latter only those cycles are excluded which are reachable by the traveling packets. In [5] Schmid et al. also show that updating as many nodes as possible is NP-hard, both under SLF and RLF.

For this thesis we will concentrate on a slightly different optimization problem. Instead of maximizing the number of nodes we update simultaneously, we will instead try to minimize the number k of rounds/subsets the controller uses to update all nodes. Intuitively, this corresponds in practice to minimizing the total time required by the update procedure. Schmid et al. [6, 7] have analyzed this problem both for the SLF and RLF consistency guarantees. While other consistency properties exist, we will limit our research to these two. For SLF, the problem is again NP-hard, while for RLF this is unknown [6, 7]. The *Peacock* algorithm achieves $\mathcal{O}(\log n)$ rounds for the RLF variant for any instance of the problem (with n nodes), and is therefore also an $\mathcal{O}(\log n)$ approximation. For SLF, there exist examples of instances that require $\Omega(n)$ rounds [7].

Our goal is to find good approximation algorithms for both the SLF and RLF Update Problems. We discuss approximation algorithms and some common techniques to devise such algorithms in Chapter 2. In Chapter 3 we apply different classical techniques and analyze their utility both from a theoretical and a practical perspective for the SLF Problem. The Linear Programming (LP) approach turns out to provide us with both an exact algorithm and an approximation algorithm whose results are most often no worse than 3 times the optimum (for randomly generated instances). In Chapter 4 we present multiple heuristics for

the RLF Problem, including the already known Peacock [5, 7] algorithm, as well as a similar LP approach to the one used in the SLF case and a Local Search algorithm. We show that Peacock is not an $o(\log n)$-approximation, closing an open question from [7], and also fix a mistake from [7]. We observe that some of the heuristics share a common idea (Section 4.1.1) and confirm through experiments that, in practice, the Local Search approach offers the most reliable approximations. The results of Local Search are in fact no worse than 2 times the optimum for all instances we tested. It remains open whether this is generally true and whether an $o(\log n)$ approximation even exists for the RLF Problem.

2 Preliminaries

2.1 Linear Programs for Approximation Algorithms

This section contains a minimal exposition of the techniques we plan to use in this thesis. For more in depth explanations we point out to [8] and [9].

2.1.1 Basics

An approximation algorithm ALG is a polynomial-time algorithm which computes a valid solution $ALG(\sigma)$ for an instance σ of a (w.l.o.g.) minimization problem $\mathcal{P}$. If $OPT(\sigma)$ is the optimum/minimum of this problem, and we can show that $ALG(\sigma) \leq C(n) \cdot OPT(\sigma)$ for all instances σ of size n, then we say that ALG is a $C(n)$-approximation of the problem [8] ($C(n)$ can be either some positive constant or some positive function depending on the size n of σ, like $\log n$.) Classical techniques to design good approximation algorithms include Greedy, Local Search, Linear Programming Relaxations, Primal-Dual Scheme, Semidefinite Relaxations etc. [8] We assume that the reader is experienced with Greedy and Local Search approaches, and shortly discuss in the following Linear Programs and their applications to approximation algorithms:

Definition 1 (Linear Program)**.** A *linear program* (LP) is an optimization problem of the form:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \ & c^T x \\ \text{s.t.} \ \ & Ax \geq b \\ & x \geq 0 \end{aligned}$$

where $c \in \mathbb{Z}^n, b \in \mathbb{Z}^m, A \in \mathbb{Z}^{m \times n}$ are the parameters of the LP. If x is also constrained to be integral (i.e. $x \in \mathbb{Z}^n$) then the program is instead called an *integer linear program* (ILP). We note that any constrained optimization problem in which both the objective function and the constraints are linear can easily be brought into the form above.

We call any x fulfilling the constraints a *(feasible) solution* of the LP/ILP. In particular, the solution x^* which achieves the *optimum* $c^T x^*$ for the LP is called *optimal solution* or *optimal fractional solution*. The optimal solution of the ILP is called *optimal integral solution*.

There is a very rich literature on LPs and ILPs [8]. The main difference between the two variants is that one can solve LPs in polynomial time, whereas solving ILPs is NP-hard. For approximation algorithms, we usually start by modeling an instance σ of the problem $\mathcal{P}$ with an ILP(σ), i.e. the solution x is required to be *integral*, and then *relax* this program to a LP(σ) by dropping off the integrality conditions on x.

Important note: ILP(σ) and LP(σ) must have a size polynomial in the size n of the instance σ. This must hold because we want to find an approximation algorithm for the problem $\mathcal{P}$, which per definition must be a polynomial-time algorithm. Therefore, we must be able to solve LP(σ) in poly(n)-time, which is possible if the size of LP(σ) is polynomial in n. In Section 2.1.3 we show that this constraint can be relaxed under some conditions.

If σ is an instance of $\mathcal{P}$, and $OPT(\sigma)$, $LP_{OPT}(\sigma)$ are the optima of the IP and (relaxed) LP respectively, then it is clear that $LP_{OPT}(\sigma) \leq OPT(\sigma)$. This is because any feasible x for the ILP is also feasible for the LP (but the converse does not hold in general). We will later see that the following quantity is important for designing approximation algorithms:

Definition 2 (Integrality Gap)**.** The maximum of:

$$\frac{OPT(\sigma)}{LP_{OPT}(\sigma)}$$

over all instances σ of some fixed size n of the problem $\mathcal{P}$ is called the *integrality gap* $\mathcal{I}(n)$ of the linear program.

Because we can only solve LPs efficiently, we need to be able to properly interpret fractional solutions. The following definition makes this idea precise:

Definition 3 (Rounding Linear Programs)**.** Let $\mathcal{P}$ be a minimization problem modeled (for an instance σ) by an ILP(σ) that has been relaxed to a LP(σ). Let $LP_{OPT}(\sigma)$ be the optimum of this LP, with corresponding solution x^*. In general, x^* is not integral.

A (polynomial-time) algorithm which, for any instance σ and corresponding optimal fractional solution $x^*(\sigma)$, maps this solution to an *integral* solution $x_I(\sigma)$ (which is feasible for the initial problem, i.e. for the ILP) is called a *rounding algorithm*. In general, the *rounded value* $x_I(\sigma)$ is not optimal for the integral program.

Now assume that we have a rounding algorithm such that the rounded value is not "much worse" than the optimal fractional solution. More precisely, there exists some $C(n)$ (either positive constant number or function depending on $n := |\sigma|$)) such that, for any instance σ of size n of $\mathcal{P}$, we have:

$$(c^T x_I(\sigma)) \leq C(n) \cdot (c^T x^*(\sigma))$$

Then, we get that:

$$(c^T x_I(\sigma)) \leq C(n) \cdot (c^T x^*(\sigma)) \leq C(n) \cdot LP_{OPT}(\sigma) \leq C(n) \cdot OPT(\sigma)$$

which proves that we have a $C(n)$-approximation algorithm for the problem $\mathcal{P}$.

We finish with the observation that, if $\mathcal{I}(n)$ is the *integrality gap* of our linear program for instances of size n, then $C(n) \geq \mathcal{I}(n)$. This shows that the approximation factor of any rounding algorithm is bounded below by the integrality gap of the underlying LP. Hence, if the integrality gap of an LP is bad, we cannot hope to get a good approximation with a rounding approach.

2.1.2 Duality

Every LP has a dual LP which is easy to compute:

Definition 4 (Dual Linear Program)**.** The *dual* of the LP from Definition 1 (called *primal*) is given by:

$$\begin{aligned} \max_{y \in \mathbb{R}^m} \ & b^T y \\ \text{s.t.} \ \ & A^T y \leq c \\ & y \geq 0 \end{aligned}$$

We call any y fulfilling the constraints a *(feasible) dual solution*. In particular, the solution y^* which achieves the *optimum* $b^T y^*$ is called *optimal dual solution*.

It can be proven that [9]:

Theorem 1 (Weak and Strong Duality)**.** *Let x^* and y^* be the optimal solutions of the primal and dual LPs respectively. Then, it holds that:*

$$b^T y^* = c^T x^*$$

i.e. the primal and the dual have the same optimum. This is called the Strong Duality Theorem. For proving approximation guarantees, Weak Duality usually suffices, i.e. we only use that $b^T y^* \leq c^T x^*$.

The above result gives us another strategy for finding good approximation algorithms. If one proves that, for any instance σ of size n of a problem $\mathcal{P}$, there exist feasible (not necessarily optimal) primal and dual solutions x and y for LP(σ) and its dual, such that x is integral and:

$$c^T x \leq C(n) \cdot (b^T y)$$

for some $C(n)$ (either positive constant number or function depending on $n := |\sigma|$)), then by Weak Duality we get that:

$$c^T x \leq C(n) \cdot (b^T y) \leq C(n) \cdot (b^T y^*) \leq C(n) \cdot (c^T x^*) \leq C(n) \cdot LP_{OPT}(\sigma) \leq C(n) \cdot OPT(\sigma)$$

which proves that we have a $C(n)$-approximation algorithm for the problem $\mathcal{P}$.

2.1.3 Ellipsoid Method and Separation Oracles

In the previous subsection we imposed the condition that the size of LP(σ) is polynomial in the size n of the instance σ of problem $\mathcal{P}$. Here we show that this condition can be sometimes relaxed. The first polynomial-time algorithm for solving LPs was the Ellipsoid method developed by Khachiyan [9]. Assuming an LP contains $m = \text{poly}(n)$ variables and exponentially many constraints (using the m variables), this algorithm works in poly(m) = poly(n) time if it has access to a *separation oracle* [8], which we now define:

Definition 5 (Separation Oracle)**.** Consider the feasible region (of an LP) determined by the inequality $Ax \leq b$, where $A \in \mathbb{Z}^{m \times n}$, $b \in \mathbb{Z}^m$ are the parameters, and m can be as large as exponential in n. We are given a fixed $x \in \mathbb{Z}^n$. A *Separation Oracle* is a polynomial-time algorithm which, for any such given x, either correctly confirms that x lies in the feasible region or correctly indicates the constraint $i \in [m]$ which x does not fulfill (i.e. such that $A_i x > b_i$). Note that, because m is at most exponential in n, the index i of this constraint can be encoded using polynomially many bits.

In later chapters of the thesis we will often define LPs with poly(n) variables (where n is the size of the problem instance σ) and exponentially many constraints. To show that these LPs are solvable in poly(n) time we provide a Separation Oracle and make use of the result presented here.

2.2 Software-defined Network (SDN) Updates

Most of the material in this section can be found in [6, 7].

In Software-defined Networks (SDN) we are given n routers/switches $V := \{1, \ldots, n\}$, which we also call *nodes*. Packets are initially sent from the *source* $s := 1$ to the *destination* $d := n$ using the path $s = 1, 2, 3, \ldots, n-1, n = d$. We want to change this path to a new path given by $\sigma(1) = 1, \sigma(2), \sigma(3), \ldots, \sigma(n-1), \sigma(n) = d$ where σ is a permutation of $\{1, 2, \ldots, n\}$. Doing this amounts to each node $1 \leq i < n$ forwarding to $\sigma(\sigma^{-1}(i) + 1)$ instead of forwarding to $i + 1$ (as in the old route). We say that a node has (been) *updated* as soon as it changes its forwarding mechanism in this way.

The nodes are updated by a central controller. In the *first round*, the controller chooses a subset $V_1 \subseteq V$ of the nodes, and instructs these nodes to update their forwarding tables according to σ. However, the controller does not know the order in which the nodes from V_1 will update. In fact, we assume that all orders are possible. It can therefore happen that some of the nodes from V_1 use their old forwarding table while others update and use the new policy. Still, after enough time, it is safe to assume that all nodes from V_1 have updated, and the controller can schedule the next round.

In general, when scheduling *round* $t \geq 2$, the controller can assume that the nodes from $V_{<t} := \cup_{j=1}^{t-1} V_j$ have already updated, and it needs to choose a subset V_t of $V \setminus V_{<t}$ to update in the current round. Again, the nodes from V_t can update in any order and the controller cannot assume any particular order. Eventually, the controller updates all nodes, and its *solution/schedule* consists of the subsets $V_1, V_2, \ldots, V_k$ it has chosen. Note that V is partitioned by these subsets, that the order in which we list the subsets is important, and that k is the total number of rounds.

In general, the controller needs to update the routers while at the same time ensuring that certain guarantees are met. For the thesis, we will mainly concentrate on two such consistency guarantees, which have been introduced in [6, 7]. The following definitions prepare and introduce these consistency notions:

Definition 6 (Full Graph)**.** For an SDN update problem with n nodes and new path/permutation σ, let $G := (V = [n], E)$ be the directed graph given by the edges $E := \{(i, i+1) : i \in [n-1]\} \cup \{(i, \sigma^{-1}(\sigma(i) + 1)) : i \in [n-1]\}$. Hence, the graph G, which we call the *Full Graph*, contains both the edges of the old and of the new path.

Definition 7 (Update Graph)**.** Assume that we have already updated the nodes from $V' \subseteq V$. This corresponds to the *Update Graph* $G_{V'} := (V, E_{V'})$ given by:

$$E_{V'} := \{(i, \sigma(\sigma^{-1}(i)+1)) : i \in V'\} \cup \{(i, i+1) : i \in V \setminus V'\}$$

i.e. the already updated nodes can only use their new edge, whereas the other nodes can only use their old edge.

We are now ready to introduce the two relevant consistency properties:

Definition 8 (Strong and Relaxed Loop Freedom (SLF))**.** For an SDN update problem with n nodes and new path/permutation σ, let $V_1, \ldots, V_k$ be a partitioning of V. The plan is to schedule in each round t the nodes from V_t. Recall the notation $V_{<t} := \cup_{j=1}^{t-1} V_j$.

We say that the schedule fulfills the *Strong Loop Freedom (SLF)* property iff, for all $t \geq 1$ and subsets $V' \subseteq V_t$, the updated subgraph $G_{V_{<t} \cup V'}$ is cycle-free. This means that, independently of the precise order in which the nodes from V_t update (recall our assumption that all orders are possible), there will never be any loop in the current update graph.

We say that the schedule fulfills the *Relaxed Loop Freedom (RLF)* property iff, for all $t \geq 1$ and subsets $V' \subseteq V_t$, the updated subgraph $G_{V_{<t} \cup V'}$ does not have any cycles reachable from the source node $s = 1$. This means that, independently of the precise order in which the nodes from V_t update (recall our assumption that all orders are possible), there will never be any loop reachable from s in the current update graph.

Relaxed Loop Freedom is usually an acceptable choice in practice, because cycles which are not reachable by any packets are not a serious issue. For both variants one can formulate the following optimization problem:

Definition 9 (SLF/RLF Optimization Problem)**.** Find a schedule $V_1, \ldots, V_k$ with a minimum number of rounds k and which fulfills the SLF/RLF consistency guarantee.

The above problem has been introduced in [6, 7] and the two variants are quite different. For SLF, Schmid et al. could show that deciding whether a valid schedule with $k = 3$ rounds exists is already NP-hard (the $k = 2$ case is easy for both variants) [6, 7]. For RLF, it is unknown whether the problem is NP-hard (for $k \geq 3$), but the *Peacock* algorithm always finds a valid (RLF) schedule with $\mathcal{O}(\log n)$ rounds, and is therefore also a $\mathcal{O}(\log n)$ approximation for the problem [6, 7]. It is an open question whether Peacock is in fact a constant-factor approximation and/or whether a better than $\mathcal{O}(\log n)$ approximation even exists. Turning

back to the SLF variant, no approximation algorithm is known better than the trivial $\mathcal{O}(n)$ algorithm which schedules one node per round.

2.3 Boxplots

For graphically representing the data obtained through our experiments, which include comparisons between the numerical results of different algorithms, we use Boxplots. In simple terms, a Boxplot is a standardized type of plot which shows five important elements of a (numerical) dataset: the minimum, the maximum, the median, the first quartile (Q_1) and third quartile (Q_3). The plot is formed of a box, which is drawn from Q_1 to Q_3 and the whiskers extend about $1.5\dot{(}Q_3 - Q_1)$ from the edges of the box. Outliers are also plotted, if they exist, as separate small dots. For more details regarding Boxplots, Wikipedia is a very good source: `https://en.wikipedia.org/wiki/Box_plot`. For our concrete plots we used the `boxplot` function from *pandas* [10, 11]. The documentation for this function can be found at: `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.boxplot.html`

3 Strong Loop Freedom (SLF) Problem

In this chapter we discuss the SLF Optimization Problem as defined in the Preliminaries (see Definition 9). Excepting the NP-hardness, which was proved for $k = 3$ rounds in [6, 7], there is little known about this problem. It is easy to prove that the schedule which updates exactly node i in round i, for all $1 \leq i \leq n-1$, is valid and uses $n-1$ rounds. Hence, we also have a trivial $\mathcal{O}(n)$ approximation algorithm. The main question is whether this upper bound can be improved.

To tackle the issue we go over some of the classical techniques in approximation algorithms and try to apply each of them to the SLF Problem. We start with two less technical methods: Greedy and Local Search, which are simple and often powerful enough to approximately solve hard scheduling problems [8]. We then move over to one of the main techniques in the field: Linear Programming Relaxations (see Chapter 2). Unfortunately, the SLF problem has an "ugly" LP relaxation which seems not to allow any direct attacks. A better way to relax the problem is to use a Parameterized Linear Program, where the number of rounds the controller can use is hard-coded instead of optimized over. While we could not prove any approximation guarantees, we implemented this Parameterized LP solver and noticed that it behaves very well in practice. It also allows to solve the problem optimally and fast for even one hundred nodes. This is the best performing implementation we are aware of for optimally solving the problem. The next sections explain the behavior of each of our approaches, both in theory and in practice.

We implemented all presented algorithms and ran simulations on an Intel i5-7200U Processor (4 cores) with 8GB RAM running Debian Stable 11 (Bullseye). Our code is publicly available at `https://github.com/RaduVintan/sdn/`. We include instructions for reproducing the results presented in this thesis. The main logic is implemented in C++. To solve linear programs we use the library Gurobi [12] in Java. We also use one specific function from the Python library Networkx [13]. More precise descriptions of our implementation are to be found in the corresponding sections.

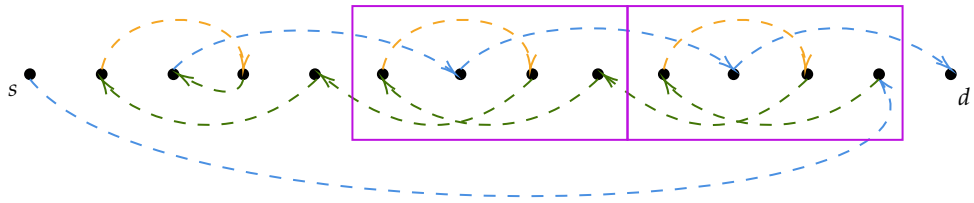


Figure 3.1: The graph ShortIsBad_2. Further purple blocks can be added.

3.1 Greedy Approach

Which nodes can we update in the first round? Consider an instance of the SLF problem with n nodes and new path/permutation σ. W.l.o.g. we can assume that the new edges $E_{\text{new}} := \{(i, \sigma(\sigma^{-1}(i)+1))\}$ and the old edges $E_{\text{old}} := \{(i, i+1)\}$ are disjoint. Consider some new edge $e := (i,j) \in E_{\text{new}}$. If $j > i$, then we call it a *forward edge*, for the obvious reason that this edge "jumps ahead" relatively to the old path. If instead $j < i$, then we call it a *backward edge* because it goes backward relatively to the old path. The node i is called *forward* or *backward node* respectively.

To showcase these new concepts, consider the instance determined by the *full graph* (recall Definition 6) in Figure 3.1. Nodes $s = 1$ to $d = 14$ are listed in the order of their old path, i.e. the i-th black dot is node i. The dashed edges indicate the new path, i.e. the permutation σ. It is now easy to see that the forward (new) edges are outgoing from the nodes $1, 2, 3, 6, 7, 10, 11$, whereas all others are backward edges. No backward node can be updated in the first round. For example, assume we update the nodes from V_1 in round 1 and $5 \in V_1$. Then, $V' := \{5\} \subseteq V_1$ and the *updated graph* (recall Definition 7) $G_{\{5\}}$ contains the cycle $5 \to 2 \to 3 \to 4 \to 5$, which shows that the SLF condition is violated (per Definition 9). One can argue in a similar way for all other backward nodes/edges.

Coming back to our question of which nodes to update in the first round, we have seen that they can only be forward nodes. It is easy to notice that updating *all* these nodes, in a greedy way, can be bad. For example, in the graph above, if these nodes have been already updated, then we have the following problem: we cannot update node 13 until 12 is already updated (else we get the cycle $10 \to 12 \to 13 \to 10$). Similarly, updating 12 requires having updated 9, which then requires 8 and so on. Our schedule will have to use one round per backward edge, leading to $1 + 6 = 7$ rounds in total (the 1 comes from the first round

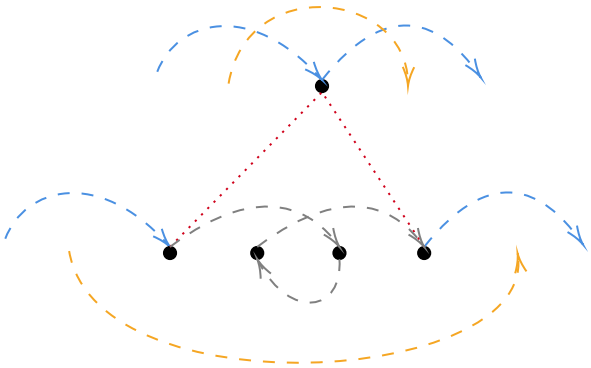


Figure 3.2: Making orange edges long and blue edges short. Only relevant nodes are shown.

where we updated all forward nodes). However, there is a schedule with 3 rounds, given by: $\{1,3,7,11\},\{4,5,8,9,12,13\},\{2,6,10\}$.

This behavior can be arbitrarily bad. Observe the two purple rectangles in Figure 3.1. The structures inside are identical, and we can repeat them arbitrarily often to obtain the graph ShortIsBad_k with k such repeating blocks. The Figure contains the graph ShortIsBad_2 with only 2 blocks. Then, similarly as in the previous paragraph, one can see that the greedy approach of updating all forward edges in the first round leads to $2k+3$ rounds, whereas the optimal schedule uses only 3 rounds.

Thus: updating all forward nodes can be dangerous. Which forward nodes should we update in the first round? Well, the trick in the above example was to update the blue forward edges first, then all backward edges, and only then the other/orange forward edges. What makes the blue forward edges "better" than the orange ones? One might think that the length is the deciding factor: the blue edges are indeed longer. This conjecture is also inspired by the *Peacock* algorithm [6, 7] for the RLF version of the problem, where long forward edges are prioritized and updated before shorter ones. However, the conjecture is in fact false for our problem. We can easily invert the roles of the blue and orange edges, making the blue edges short and the orange edges long (except the first of each), by replacing the nodes $7,11,\ldots$ with gadgets as indicated in Figure 3.2.

3.1.1 Highest Lower Bound (HLB) Algorithm

We have noticed that an instance of the SLF Problem can require a high number of rounds whenever there is a chain of dependencies between edges, like we have seen for the graph in Figure 3.2 after we updated all forward edges in the first round. In [7] the authors also show other such examples, where updating v_1 requires having already updated v_2, which in turn requires having updated v_3 etc. In this section we present an algorithm which aims to give us a hint about how many rounds might be required.

We again consider the *full graph* G determined by an instance (n, σ) of the SLF Problem. Every forward edge can be updated immediately in the first round. A backward edge (i, j) connects node i to node $j < i$. Before updating it, at least one of the nodes $i+1, \ldots, j-1$ must have already been updated, because otherwise we get the cycle $j \to i \to i+1 \to \cdots \to j$. But, assuming for example that all these intermediary nodes are also backward nodes, updating (i, j) will not be only possible until at least round 3. This observation motivates Algorithm 1, which for every node i computes a *lower bound* `lb[`i`]` for the number of rounds we need to wait until updating i, under the assumption that the *forward nodes* in the set S were already updated. We prove this in the following:

Lemma 1. *Let $S \subset V$ be a subset of forward nodes and $HLB(S)$ be the output of the HLB Algorithm ran on the input S. Let R be the number of rounds achieved by a valid schedule which updates all nodes in S (and maybe others) in the first round. Then $HLB(S) \leq R$.*

Proof. Let $R[i]$ be the round in which the given valid schedule updates node i. Because $HLB(S) = \max\{\texttt{lb}\}$, it is sufficient to prove that, at the end of the execution of the algorithm, we have $\texttt{lb}[i] \leq R[i]$ for any i. Because $R[i] \geq 1$ for any i, this easily follows for all forward nodes i and for all backward nodes i for which Line 7 is reached. For the backward nodes i which reach Line 10, this is also clear because $i \in P_j$ implies that $R[i] \geq 2$. We are left with the backward nodes i for which $i \in P_j$ and no forward non-updated node k appears in the path P_j before i. We call such nodes i *interesting*.

We first show that for any *interesting* node i there is a $k \in K_i$, s.t. $R[k] < R[i]$. (This also implies that K_i is non-empty). Assume the contrary and consider the round $R[i]$. We consider the case where i updates before any node in K_i. Notice then that during this round there is a cycle C formed by concatenating the backward edge (i, j) with the path P_j capped at i. Because i is interesting and no node from K_i was updated, all nodes in this path are either

Algorithm 1: Highest Lower Bound (HLB) Algorithm

Input: S: set of already updated nodes, all forward
`lb`: vector of size n, set to 1 for forward nodes
for $i = 1$ to $n - 1$ and i backward **do**
    $j \leftarrow$ node the (backward) edge from i leads to
    P_j : path from j to d by taking currently activated edges (determined by S)
    if $i \notin P_j$ **then**
        $\texttt{lb}[i] \leftarrow 1$
        continue for loop
    if $i \in P_j$ and $\exists\, k \in P_j$, s.t. k is a non-updated ($k \notin S$) forward node appearing before i in P_j **then**
        $\texttt{lb}[i] \leftarrow 2$
        continue for loop
    $K_i := \{k \in P_j : j \leq k < i,\ j \text{ backward}\}$
    $\texttt{lb}[i] \leftarrow 1 + \min\{\texttt{lb}[k] : k \in K_i\}$
end
return $\max\{\texttt{lb}\}$

forward updated nodes or backward non-updated nodes from K_i, and hence C contains exclusively edges that currently exist, contradiction.

Now we prove $\texttt{lb}[i] \leq R[i]$ for all *interesting* nodes i per induction on their index. Let i be the first interesting node. By the previous paragraph, there exists $k \in K_i$, s.t. $R[k] < R[i]$ and k is not interesting (because i is the first one). But then we already know that $\texttt{lb}[k] \leq R[k]$, and thus:

$$\texttt{lb}[i] \leq 1 + \texttt{lb}[k] \leq 1 + R[k] \leq R[i]$$

If i is an arbitrary interesting node then k might also be interesting, but then we get that $\texttt{lb}[k] \leq R[k]$ by the induction hypothesis and we can continue as above. This settles the proof. □

Corollary 2. *Let $HLB(\varnothing)$ be the output of the HLB Algorithm produced for the input $S = \varnothing$ and OPT be the optimal number of rounds achievable by any valid SLF solution. Then $HLB(\varnothing) \leq OPT$.*

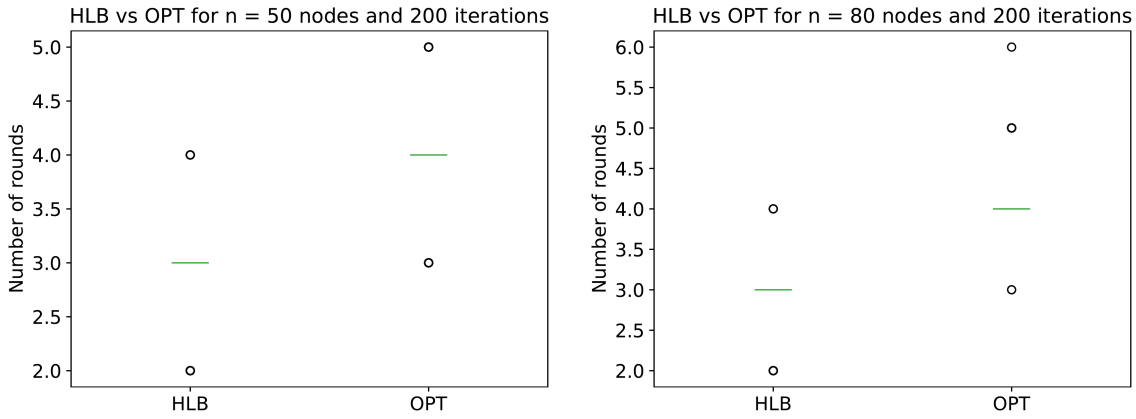


Figure 3.3: HLB Experiments

We computed the values $HLB(\varnothing)$ and OPT for different randomly generated permutations in order to validate the previous Corollary. The distribution of the results is shown in the Boxplots at Figure 3.3. (Details regarding the computation of OPT are available in Section 3.3.2.) As expected, the values produced by the HLB algorithm are smaller than the true optima, and increasing the number of nodes n seems to increase the difference between the two types of quantities. It is also apparent that $HLB(\varnothing)$ and OPT measure 3 and 4 rounds respectively for most permutations.

Algorithm 2: HLB Heuristic

```
1 S ← set of all forward nodes
2 HLB ← HLB(S)
3 for i = 1 to n − 1 and i forward do
4 |   if HLB > HLB(S \ {i}) then
5 |   |   HLB ← HLB(S \ {i})
6 |   |   S ← S \ {i}
7 end
8 return S
```

Algorithm 1 allows us to specify any set S of already updated forward nodes. This flexibility can be exploited to produce an algorithm for finding a good choice of forward edges to

```
Highest lower bound is: 6
Highest lower bound without forward edge 2 is: 5
Gave up on forward edge 2
Highest lower bound without forward edge 3 is: 6
Highest lower bound without forward edge 6 is: 3
Gave up on forward edge 6
Highest lower bound without forward edge 7 is: 4
Highest lower bound without forward edge 10 is: 1
Gave up on forward edge 10
Highest lower bound without forward edge 11 is: 2
```

Figure 3.4: Output of HLB Heuristic for ShortIsBad_2

update in the first round. Our HLB based heuristic is presented as Algorithm 2. It starts by computing the HLB under the assumption that we update *all* forward nodes in the first round. Then, it goes over all forward nodes i and computes the HLB assuming that i is no longer updated. If this leads to an improvement of the HLB, i is taken out of the set S. The resulting set S of forward nodes that can be safely updated is returned. It is interesting to look at the output of the Algorithm on the ShortIsBad_2 graph shown in Figure 3.1. We already showed for this graph that updating all forward nodes in the first round leads to a severely sub-optimal solution. The output of Algorithm 2 for this instance is shown in Figure 3.4. The Algorithm correctly decides to avoid updating the nodes 2, 6 and 10.

Unfortunately, it is not clear at all how one can extend this idea to continue finding suitable updates in the following rounds. Moreover, we have not been able to prove any guarantees regarding the highest lower bound computation from Algorithm 2, so proving any approximation guarantee seems not to be possible. We therefore move on to other approaches.

3.2 Local Search

Local Search algorithms start with a simple feasible solution of an optimization problem and then continually try to improve the solution, using a local criterion, until no more improvements can be found. They are designed to run in polynomial time. Then, an approximation factor is proven usually by showing that a locally optimal solution cannot be much worse than a globally optimal solution. Sometimes this follows trivially, for example if we are optimizing over a convex region; other times a proof is hard to find and local search remains just a heuristic. For more details and other applications we point out to [8].

How would a Local Search approach look like for our SLF Problem? Firstly, we need to

understand how to check efficiently whether a proposed solution is feasible. This is essential for any Local Search approach: we cannot change our initial solution if we cannot check efficiently that a neighboring solution is feasible. Also note that we cannot directly use Definition 9. That would require checking for all subsets of $V' \subseteq V_t$ that no cycles exist in the *update graph* (recall Definition 7) $G_{V_{<t} \cup V'}$. But we can have $|V_t| = \Omega(n)$ and then we would have to check exponentially many subsets. We therefore need to simplify this criterion. In order to achieve this, we introduce a more general notion of update graphs:

Definition 10 (Refined Update Graphs)**.** Assume that we have already updated the nodes from $V' \subseteq V$ and plan to update the nodes from V'' in the next round. This corresponds to the *Refined Update Graph* $G_{V',V''} := (V, E_{V',V''})$ given by:

$$E_{V',V''} := \{(i, \sigma(\sigma^{-1}(i)+1)) : i \in V' \cup V''\} \cup \{(i, i+1) : i \in V \setminus V'\}$$

i.e. the already updated nodes can only use their new edge, the nodes we update in the current round can use both their new and old edges, and the other nodes can only use their old edge.

We show the following:

Lemma 3. *Let $V_1, \ldots, V_k$ be a schedule whose (SLF) feasibility we want to check. We are in round t, i.e. $V_{<t}$ were already updated and we update the nodes from V_t. Then: the update graph $G_{V_{<t} \cup V'}$ is cycle-free for all subsets $V' \subseteq V_t$ iff the refined update graph $G_{V_{<t},V_t}$ is cycle-free.*

Proof. If $G_{V_{<t} \cup V'}$ contains a cycle for some $V' \subseteq V_t$, then this cycle is made up of new edges outgoing from nodes in $V_{<t} \cup V'$ and old edges outgoing from nodes in $V \setminus (V_{<t} \cup V')$. All these edges also exist in $G_{V_{<t},V_t}$ which therefore contains a cycle.

Now assume $G_{V_{<t},V_t}$ contains a cycle C. We identify a cycle with the vertices it contains, i.e. $C \subseteq [n]$. We define the sets C_{old} and C_{new} depending on whether the outgoing edge of a vertex $i \in C$ is old or not. Note that C_{old} and C_{new} thereby partition C. Consider the update graph $G_{V_{<t} \cup C_{\text{old}}}$. It is easy to check that C is also a cycle in this graph, which finished the proof. □

The above lemma gives a very simple polynomial-time algorithm to check the validity of a schedule $V_1, \ldots, V_k$. We just need to check for all $1 \leq t \leq k$ that the graph $G_{V_{<t},V_t}$ is cycle-free, which can be done by adapting the well-known Depth-First-Search (DFS) traversal.

To design the Local Search algorithm we still need to define formally what will count as a neighboring solution. First, let us identify any (feasible) solution with the list $(V_1, \ldots, V_k)$ where, as always, V_t consists of the nodes we update in round t.

Definition 11 (2-Neighborhood)**.** Let $s := (V_1, \ldots, V_k)$ be a valid schedule. Then, the *2-neighborhood* $N_2(s)$ of s is the set of *feasible* schedules $s' := (V'_1, \ldots, V'_k)$, such that s' is obtained from s by choosing two nodes $u \in V_i$, $w \in V_j$ (for any i, j) and moving them from V_i and V_j to subsets $V_{i'}$ and $V_{j'}$ (for any i', j') respectively. Hereby i, j, i', j' are indices from 1 to k. They are not required to be distinct, which implies, in particular, that $s \in N_2(s)$ and that $N_2(s)$ also contains the schedules s' obtained from s by moving just one node from one subset to another ($N_1(s) \subseteq N_2(s)$).

One can analogously define k-neighborhoods for any $k \geq 1$. Working with 2-neighborhoods will prove to be enough to reach the important conclusions regarding Local Search. Note that in the above definition the schedule $s' := (V'_1, \ldots, V'_k)$ might contain empty subsets. This can happen if, for example, s' is reached from s by moving the only element $u \in V_i$ to some other subset $V_{i'}$. Hence, in the context of this section we will assume that any schedule $s := (V_1 \ldots, V_k)$ may contain empty subsets, and define the *number of rounds of* s as $r(s) := |\{i \in [k] : V_i \neq \varnothing\}|$. We also define $\min(s) := \min\{|V_i| : V_i \neq \varnothing\} \geq 1$.

We are now ready to present our Local Search algorithm:

Algorithm 3: Local Search with 2-Neighborhood

```
s initial schedule with r(s) = k rounds
while true do
    if ∃s' ∈ N_2(s) with r(s') < r(s) then
        s ← s'
    else if ∃s' ∈ N_2(s) with min(s') < min(s) then
        s ← s'
    else
        return s
    end
end
```

Why is the if statement at Line 5 helpful? Well, if we only try to find schedules s' with $r(s') < r(s)$ we are locally optimal as soon as all non-empty subsets V_i have size $|V_i| \geq 3$. In this case it trivially holds that $\nexists\, s' \in N_2(s)$ with $r(s') < r(s)$. On the other hand this case is quite common for instances where n is big and the number of rounds one can reach is constantly small. Therefore, if all non-empty subsets are large, we instead try to decrease the smallest such subset, in the hope that we will eventually reach a state where again the number of non-empty subsets can be decreased. Even with this addition, it is easy to see that Algorithm 3 terminates and returns a feasible schedule which is locally optimal.

3.2.1 Experiments

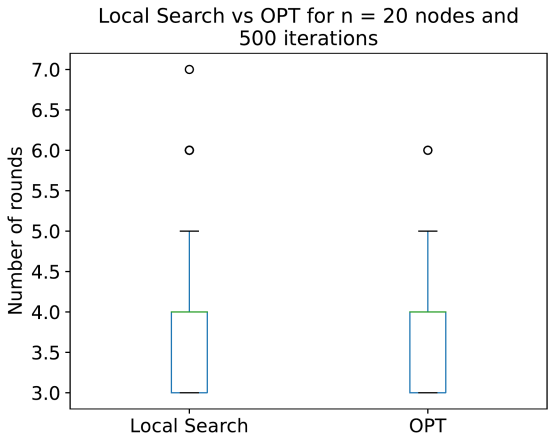
We compare the results obtained by our (C++) implementation of Local Search with the optimum. We compute the optimum by using the improved LP Relaxation from Section 3.3.2. The results can be observed in Figures 3.5a, 3.5b and 3.5c. We notice that the Local Search solution is usually smaller than $2 \cdot OPT$. The algorithm seems to provide reasonable results even for larger n, though a small decrease in the approximation quality is apparent.

Unfortunately, while the algorithm is quite good in practice, there are graph families for which the approximation factor can be arbitrarily bad. One such is given by ShortIsBad$_k$ graphs introduced in Section 3.1. Each such graph has $n = 4k + 2$ nodes and can be scheduled in 3 rounds. Running the Local Search routine for different k leads to the results shown in Table 3.1.

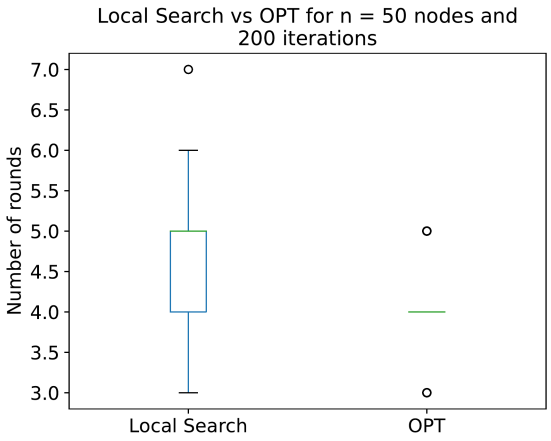
k	Number of rounds
5	3 Rounds
8	5 Rounds
10	6 Rounds
15	7 Rounds
22	8 Rounds

Table 3.1: Running Local Search on ShortIsBad$_k$ ($n = 4k + 2$ nodes)

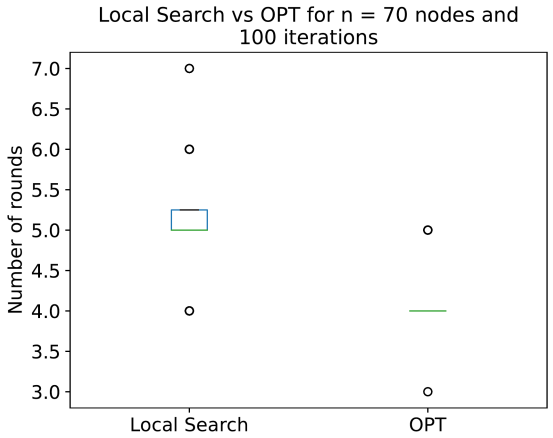
These results suggest that the number of rounds Local Search uses will tend to infinity as k increases arbitrarily. We do not expect that choosing bigger neighborhoods or starting with different initial schedules in our Local Search routine can solve this problem.



(a) Maximum difference: 2, Maximum approximation factor: 1.66, Mean difference: 0.148, Mean approximation factor: 1.041



(b) Maximum difference: 3, Maximum approximation factor: 2.0, Mean difference: 0.665, Mean approximation factor: 1.169



(c) Maximum difference: 3, Maximum approximation factor: 1.75, Mean difference: 1.01, Mean approximation factor: 1.252

Figure 3.5: Local Search Experiments

3.3 Linear Programming (LP) Relaxation

3.3.1 First Approach

Let $G = ([n], E)$ be the graph induced by an instance (n, σ) of the SLF Problem, i.e. $E := \{(i, i+1) : i \in [n-1]\} \cup \{(i, \sigma^{-1}(\sigma(i)+1)) : i \in [n-1]\}$ and thus the graph G contains both the edges of the old and of the new path. We call these edges *old* and *new* edges respectively. W.l.o.g. we can and will assume that the sets E_{old} and E_{new} of old and new edges are disjoint (and partition E).

Our LP relaxation will impose the SLF property. To write the corresponding conditions compactly, we require some notation. Let $\mathcal{C}$ be the set of the cycles of G and consider some fixed cycle $C \in \mathcal{C}$. We identify a cycle with the vertices it contains, i.e. $C \subseteq [n]$. We define the sets C_{old} and C_{new} depending on whether the outgoing edge of a vertex $i \in C$ is old or not. Note that C_{old} and C_{new} thereby partition C. (This notation also appeared in the proof of Lemma 3)

We now show the following:

Lemma 4. *Define the variables y_{ti} for all $t \in \{0, \dots, n\}$ and $i \in [n]$ and consider the following LP:*

$$\begin{aligned} \min \sum_{t=1}^{n} z_t & \\ \text{s.t. } y_{0i} &= 1, \quad \forall i \in [n] \\ y_{ni} &= 0, \quad \forall i \in [n] \\ x_{ti} + y_{t-1,i} &= 1, \quad \forall t, i \in [n] \\ y_{t-1,i} - y_{ti} &\geq 0, \quad \forall t, i \in [n] \\ z_t - y_{t-1,i} &\geq 0, \quad \forall t, i \in [n] \\ \sum_{i \in C_{old}} x_{ti} + \sum_{y \in C_{new}} y_{ti} &\geq 1, \quad \forall t \in [T], \forall C \in \mathcal{C} \end{aligned}$$

This LP is a relaxation of the SLF Problem. More precisely, every feasible solution of the SLF instance can be encoded as an integral solution of the LP, and the optimal integral solution of the LP corresponds to the optimal solution of the SLF instance.

Proof. Understanding why the lemma holds is trivial once we build some intuition for how the variables are used and what they signify. First, note that $0 \leq y_{ti}, x_{ti} \leq 1$ for any t, i, and hence any integral solution is binary. $y_{ti} = 1$ should hold iff at time t the vertex i was *not yet* updated and, thus, the new edge cannot be used (yet). This explains why we require

$y_{0i} = 1$ (i.e. before starting the algorithm the routers are not yet updated and we use the old path) and $y_{ni} = 0$ (i.e. at the end all routers should be updated and we use the new path). $x_{ti} = 1$ holds iff the old edge can no longer be used. To see this, note that $x_{ti} = 1 - y_{t-1,i}$, which implies that: if at time $t-1$ vertex i was not still updated, then its old edge is still be usable at least until time $t+1$, so in particular also during the t-th round (so that $x_{ti} = 0$). On the contrary, if vertex i was updated in a round $\leq t-1$, then the old edge is no longer usable (so that $x_{ti} = 1$). The monotonicity constraints on y ensure that updating a vertex is irreversible, and the cycle constraints ensure that the update policy fulfills the SLF condition. In an optimal integral solution, $z_t = 1$ iff round t was required and therefore this solution minimizes the number of rounds. □

To find a feasible *fractional* solution (or show that no such solution exists) in polynomial time, we can use the Ellipsoid method with a suitable separation oracle for the cycle breaking constraints. To check whether any such constraint is violated for a given instantiation of the variables y and x in polynomial time, for any time $t \in [n]$, we find the shortest cycle in G, using the x_{ti}-s as weights for the old edges and the y_{ti}-s as weights for the new edges, and return this cycle iff it has length < 1. If we cannot find such a cycle we are done. Finding shortest cycles (in a weighted directed graph without negative cycles, like in our case) can be done by adapting the well-known Floyd-Warshall algorithm.

Problems of the LP Relaxation

The plan is to design a suitable rounding algorithm for the above LP relaxation. Using the well known LP solver Gurobi [12] in Java we generated and solved the relaxation for different instances of the SLF Problem. Writing the LP down requires generating all cycles of the *full graph* G of the instance. For that, we used the function `simple_cycles` from the networkx [13] Python library which conveniently lists all cycles of any directed graph.

However, (fractionally) solving the LP for the ShortIsBad$_k$ instances reveals some unpleasant behavior, which is presented in Table 3.2. Let y^* be the optimal solution of the LP (for some instance of the SLF Problem). Imagine the values of y^* listed as a $(n+1) \times n$ matrix Y^*, where the rows match to time/rounds, and the columns match to vertices. We denote with y_t^* the vectors $(y_{ti}^*)_{i \in [n]} \in \mathbb{R}^n$, i.e. the rows of Y^*. In principle, the rows of this matrix should "become" 0 as soon as possible, i.e. the minimal t such that $y_t^* = 0$ should not be larger than

	k = 2	k = 3	k = 4	k = 5	k = 6
$y_{0,n-1}$	1.0	1.0	1.0	1.0	1.0
$y_{1,n-1}$	1.0	1.0	1.0	1.0	1.0
$y_{2,n-1}$	0.5	0.5	0.5	0.5	0.5
$y_{3,n-1}$	0.25	0.25	0.25	0.25	0.25
$y_{4,n-1}$	0.125	0.125	0.125	0.125	0.125
$y_{5,n-1}$	0.0625	0.0625	0.0625	0.0625	0.0625
$y_{6,n-1}$	0.0	0.3125	0.3125	0.03125	0.03125
$y_{7,n-1}$		0.015625	0.015625	0.015625	0.015625
$y_{8,n-1}$		0.0	0.0078125	0.0078125	0.0078125
$y_{9,n-1}$			0.0	0.00390625	0.00390625
$y_{10,n-1}$				0.00195313	0.00195313
$y_{11,n-1}$				0.0	0.00097656
$y_{12,n-1}$					0.00048828
$y_{13,n-1}$					0.0

Table 3.2: Behavior of optimal $y_{t,n-1}$ with 8 digits precision for ShortIsBad$_k$ graphs and initial LP Relaxation.

OPT (the integral optimum). This would mean that the LP correctly detects that all nodes ought to be updated as fast as possible. Unfortunately, our model does not directly capture this intuition: the LP just minimizes the sum of the maxima of each row, which can lead to pathological examples such as the one in Table 3.2. Here, even though the ShortIsBad$_k$ graphs can be updated in 3 rounds (integral optimum), our optimal LP solution, for some vertices i, uses fractional values for a much larger amount of rounds, meaning that the rows of the matrix converge to 0 very slowly. We suspect that there is no good rounding procedure for such cases. In the next section we use a different relaxation which solves these issues.

3.3.2 Improving the LP Relaxation

As we have seen, the LP relaxation from the previous chapter has some disadvantages. In this section we show another approach which makes the rows y_t^* of our fractional solutions converge to 0 as fast as possible. This is made precise by the following:

Lemma 5. *Let $T \geq 2$ be a natural number. We define the variables y_{ti} for all $t \in \{0, \ldots, T\}$ and*

$i \in [n]$. Consider the polytope $LP(T)$ given by the following set of constraints:

$$\begin{aligned} y_{0i} &= 1, \quad \forall i \in [n] \\ y_{Ti} &= 0, \quad \forall i \in [n] \\ x_{ti} + y_{t-1,i} &= 1, \quad \forall t \in [T], i \in [n] \\ y_{t-1,i} - y_{ti} &\geq 0, \quad \forall t \in [T], i \in [n] \\ \sum_{i \in C_{old}} x_{ti} + \sum_{y \in C_{new}} y_{ti} &\geq 1, \quad \forall t \in [T], \forall C \in \mathcal{C} \end{aligned}$$

Then, the constraints can be satisfied by integral variables iff there is a schedule with T rounds for the corresponding instance (n, σ) of the SLF problem.

The proof of this Lemma closely resembles the proof for Lemma 4 and is left to the reader. To find a feasible fractional point in in the polytope $LP(T)$ (for some T) or show that no such point exists (in polynomial-time) we can again adapt the Ellipsoid method using a suitable separation oracle and find short cycles using the Floyd-Warshall algorithm. The LP relaxation finds a minimal $T \geq 2$, such that the polytope $LP(T)$ introduced in Lemma 5 is feasible. This can also be done in poly-time, because for $T = n$ the polytope is (even integrally) feasible. This follows because any instance (n, σ) of the SLF problem can be solved in less than n rounds.

This relaxation helps solve the problem of slow convergence of rows we noticed in Table 3.2. In fact, the solution computed for the ShortIsBad$_k$ graphs is *always* integral. In particular, Table 3.3 shows the behavior of the variables $y^*_{t,n-1}$ for the improved relaxation.

As a final note, observe that the polytope $LP(T)$ can naturally also be used to obtain the optimal *integral* solution OPT, i.e. the optimal schedule for the corresponding SLF instance. To do that, we just have to search for *integral* feasible points inside $LP(T)$ instead of searching for *fractional* feasible points. (In other words, we impose that x_{ti} and y_{ti} are integral.) While this is no longer possible in polynomial-time, LP solvers like Gurobi [12] can find such points very efficiently in practice. As seen later in Section 4.2.4, this technique allows us to compute OPT for instances with a significant number of nodes much faster than with any other known technique.

Still, the question of finding a polynomial-time rounding procedure, which rounds optimal *fractional* solutions of the above LP, remains of interest, and we discuss a possible approach in the next section.

	k = 2	k = 3	k = 4	k = 5	k = 6
$y_{0,n-1}$	1.0	1.0	1.0	1.0	1.0
$y_{1,n-1}$	1.0	1.0	1.0	1.0	1.0
$y_{2,n-1}$	1.0	1.0	1.0	1.0	1.0
$y_{3,n-1}$	0.0	0.0	0.0	0.0	0.0

Table 3.3: Behaviour of $y_{t,n-1}$ for ShortIsBad$_k$ graphs and improved LP Relaxation. Compare with Table 3.2.

3.3.3 Rounding the LP Relaxation

Assuming we obtained an optimal feasible (fractional) solution of $LP(T)$ for a minimal T, the next question is how to round this solution to make it integral in polynomial time. Let OPT be the optimal/minimal number of rounds obtained by an integral solution and note that the (minimal) T fulfills the inequality $T \leq OPT$. This follows because we relaxed the integral problem. The rounding procedure might increase T and lead to an (integral) solution requiring $T' \geq T$ rounds. Ideally, one could show that T' is not much bigger than T, e.g. $T' \leq k \cdot T \leq k \cdot OPT$ for some constant k, which would then prove that the algorithm is a k-approximation of the SLF problem.

So, fix some minimal $T \geq 2$ for which we find a (fractional) feasible solution y^* (w.l.o.g. we can ignore x). Similar to before, imagine the values of y^* listed as a $(T+1) \times n$ matrix Y^*, where the rows match to time/rounds, and the columns match to vertices. We denote with y^*_t the vectors $(y^*_{ti})_{i \in [n]} \in \mathbb{R}^n$. i.e. the rows of Y^*. Let $2 \leq k \leq T$; we keep the following invariant for the algorithm: the rows $1, \ldots, k-1$ are already integral. Observe that, if $k = T$, then the solution is integral (because the last row of Y is made out of 0-s) and we are done. Let us concentrate on the (potentially un-integral) row y^*_k. The following definition explains how rounding works:

Definition 12. We say that we *integralize* the row k using the vector $y^*_k \in \mathbb{R}^n$ if we restrict the polytope $LP(T)$ by adding the conditions:

- $y_{ki} = 1$ if $y^*_{ki} > 0$
- $y_{ki} = 0$ if $y^*_{ki} = 0$

and search for another feasible solution (fulfilling these integrality conditions) in the restricted $LP(T)$. If we cannot find such a solution, we increase T (without removing any of the integrality conditions added by any integralizations) and retry finding a feasible solution.

With this terminology, we show the following:

Lemma 6. *Integralizing row k using y_k^* provides a feasible solution for the (restricted) polytope $LP(T')$ where $T' = T$ or $T' = T + 1$.*

Proof. Let z_k be the integral version of y_k^*. (If y_k^* is already integral, then $z_k = y_k^*$.) Then, it not hard to see that the solution given by the vectors $y_0^*, \ldots, y_{k-1}^*, z_k, y_k^*, \ldots, y_T^*$ fulfills all conditions of $LP(T+1)$ (including the conditions added by integralizing the row), which shows that $LP(T+1)$ is non-empty. □

This motivates the following algorithm:

Algorithm 4: Rounding Algorithm

```
1 Find minimal T ≥ 2, s.t. LP(T) contains feasible y*.
2 k ← 2; Loop Invariant: rows 1,...,k−1 integral.
3 while k ≤ T do
4 |   Integralize row k using y_k^* and obtain (y*)_new ∈ LP(T') for T' ∈ {T, T+1}
5 |   y* ← y*_new, T ← T', k ← k+1
6 |   if y* integral then
7 |   |   return y*
8 end
```

Definition 13. For any current solution y^* of $LP(T)$ and index k we call the vector $y_k^* \in \mathbb{R}^n$ *degenerate* if it does not contain any zeros.

If at some point the algorithm integralizes row k using a degenerate y_k^*, we might run into trouble. The newly found solution might be the one provided by the proof of Lemma 6, but then in the next round we integralize row $k+1$ using again y_k^*, leading to another duplication of this row. Thus the program does not terminate. The problem can be fixed in this case if we

manually find a vertex to update and add this update as a further constraint to the polytope, ensuring progress. However, we never encountered such a case in any of our experiments and we strongly suspect that this scenario is impossible. In fact, we conjecture the following property:

Conjecture. For any current solution y^*, and for all $1 \leq t \leq T$, there is some i such that $y^*_{ti} = 0$ but $y^*_{t-1,i} \neq 0$.

The conjecture implies that the fix discussed above is never required. We are guaranteed, in each iteration of the While-loop, to increase the number of columns which reach 0 before the last round in the current solution y^*, which in turn implies termination of the algorithm (in at most n rounds). Coming back to our discussion in the first paragraph of the section, let T be the value found in row 1 of the Algorithm (such that $T \leq OPT$), and T' be the value of the same variable at the end of the execution. If we could show that $T' \leq k \cdot T$, because $T \leq OPT$, we would obtain a k-approximation algorithm.

Implementation Details

Computing the rounded solution once the fractional optimal solution becomes available is straightforward. To determine the optimal fractional solution, we initially proceeded as suggested in the previous section: we generate all cycles of the *full graph* G of the instance using the `simple_cycles` function from the networkx [13] library, which allows us to specify the $LP(T)$ polytope (for any T), and then proceed by iteratively searching for feasible solutions in the $LP(T)$ polytope starting with $T = 1, 2, \ldots$ This approach is quite slow and even impractical for bigger n, because we need to generate exponentially many cycles to specify the polytope $LP(T)$ in Gurobi.

A different approach turns out to be much faster (in practice). We initially add no cycle constraints at all, then search for a (either fractional or integral, depending on what is computed) feasible point inside this relaxed version of $LP(T)$. If no such point exists, then adding further cycle constraints would obviously keep the polytope empty, and therefore we need as before to increase T. If instead a feasible solution x, y is found, we use the Floyd-Warshall algorithm to find the shortest cycle (w.r.t. weights x_{ti} and y_{ti}). If this cycle has length ≥ 1, we are done, because it implies that all cycles in the *full graph* G fulfill the cycle constraint. If instead the cycle is too short, we proceed by adding the corresponding constraint in Gurobi.

It is well known that Floyd-Warshall can be easily used to also determine, for each vertex $i \in [n]$, the shortest cycle containing i. We use this extension of Floyd-Warshall to add all the constraints corresponding to these cycles. We noticed that this approach of lazily adding constraints leads to very good practical performance.

3.3.4 Experiments

The statistical data gathered during our experiments is visualized in Figures 3.6 and 3.7.

In Figure 3.7 we compare the three quantities LP_{OPT}, i.e. the minimal T for which the polytope $LP(T)$ is (fractionally) feasible, OPT, i.e. the optimal/minimal number of rounds for an integral solution, and the solution ALG provided by the algorithm, i.e. the number of rounds the integral solution y^* produced by Algorithm 4 uses. These Boxplots have been produced by running our algorithms on randomly generated permutations/instances of the SLF Problem. We notice that LP_{OPT} and OPT are usually very close to each other. Indeed, cases where these two quantities are not equal are relatively rare. Also, for most instances OPT and LP_{OPT} stabilize at around 4 rounds. The rounded solution ALG is usually close to OPT, but the empirical approximation factor ALG/OPT seems to get progressively worse, both in mode and in mean, as we increase the size n of our instances. While it is in principle possible that for higher n the approximation factor stabilizes/converges to a constant, we suspect that our rounding procedure can behave poorly in pathological cases, but is usually competitive.

In Figure 3.6 we show how the mode and the mean of the empirical approximation factor behave for different values of n. The data has been produced by running 50 simulations on randomly generated permutations/instances of the SLF Problem for each $n = 60, 65, \ldots, 110$. We kept the number of simulations relatively small for performance reasons. This leads to quite high variance. In particular, there is an outlying maximal approximation factor 6.5 obtained for $n = 85$. Such spikes are relatively common for the SLF Problem: most randomly generated instances are well approximated, and instances for which the approximation factor is high are quite rare. The mean of the approximation factor seems to increase very slowly.

Comparison with another solver

In [7] a repository is provided which includes an ILP solver for different variants of the SDN Network update problem, including (but not limited to) the SLF and RLF Problems. The ILP

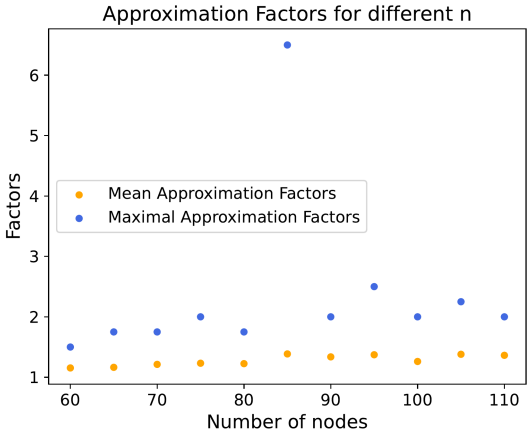
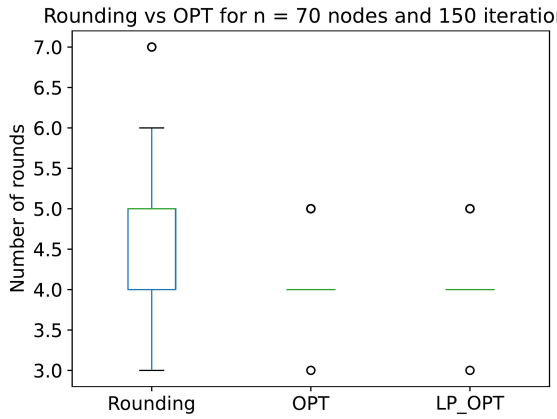


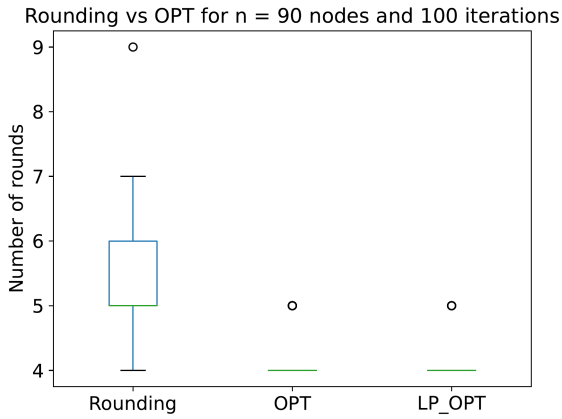
Figure 3.6: SLF: Empirical approximation performance for n between 60 and 110

is adapted from [14]. It can be used to determine the (integral) optimum OPT for any instance of these problems, and we used it to test the correctness of our implementation. As far as we know, this is the only publicly available algorithm based on Linear Programming which precedes our techniques. Our program for SLF runs all experiments required for Figure 3.7a (70 nodes and 150 permutations) in 3 minutes and 21 seconds, while the other ILP solver requires 3 minutes and 32 seconds just for the first 15 permutations. This is in a way not very surprising: the ILP from [7] is much more complex, as it supports a broader selection of problems. Hence, we conclude that our algorithm is very useful in practice for computing the optimum for instances of the SLF Problem. It also supports computing (seemingly) decent approximations if more speed is required or if the instances are larger.

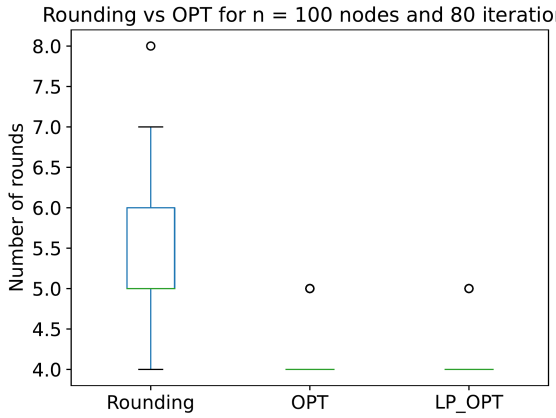
Other solvers exist based on other techniques, like Petri Networks, Linear Temporal Logic or Stackelberg Games. These solvers are again more general than our algorithms: they solve many variants of the SDN update problem which are out of scope for the thesis. Examples of such tools include the recent Kaki [15], or Netstack [16], Latte [17] and Netsynth [18]. We intended to make performance comparisons with Kaki [15], as it improves on the other methods, but unfortunately it cannot be used to solve the SLF Problem. While it can solve the RLF Problem in principle, it was not optimized for this problem specifically and cannot solve it for instances with many nodes.



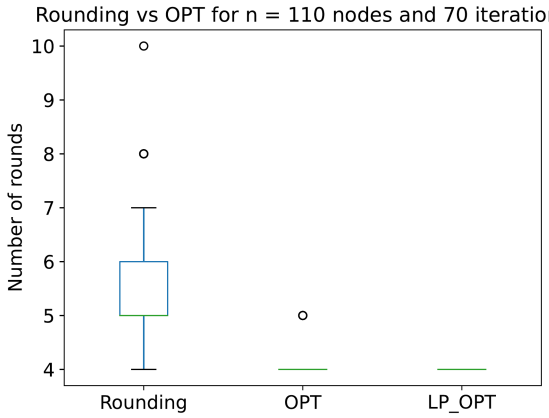
(a) Maximum difference: 3, Maximum approximation factor: 1.75, Mean difference: 0.83, Mean approximation factor: 1.206



(b) Maximum difference: 5, Maximum approximation factor: 2.25, Mean difference: 1.2, Mean approximation factor: 1.296



(c) Maximum difference: 4, Maximum approximation factor: 2.0, Mean difference: 1.3125, Mean approximation factor: 1.325



(d) Maximum difference: 6, Maximum approximation factor: 2.5, Mean difference: 1.657, Mean approximation factor: 1.412

Figure 3.7: SLF: Comparison of Rounding with OPT

4 Relaxed Loop Freedom (RLF) Problem

In this chapter we discuss the RLF Optimization Problem as defined in the Preliminaries (see Definition 9). In Section 4.1 we present the Peacock algorithm introduced in [6, 7], which provably solves any RLF instance using $\mathcal{O}(\log n)$ rounds. We also provide a framework which allows an elegant unified perspective on Peacock and some of our later algorithms. In [7] the authors provide a family of graphs to prove that Peacock can require $\Omega(\log n)$ rounds. We show that their construction is not fully correct, but that it can easily be fixed to keep the arguments intact. Further, we investigate the open question whether Peacock is an $o(\log n)$-approximation and prove by counterexample that the answer is negative. Hence, it remains open whether there exists an $o(\log n)$-approximation for the RLF Problem.

Similarly to the previous chapter, in Section 4.2 we present several heuristics which behave very well in practice for the RLF Problem. In particular, the LP approach (inspired from the previous chapter) also allows us to compute the optimum in a fast manner even for instances with a significant number of nodes. As far as we are aware, this is the best performing implementation available for optimally solving the RLF Problem. The Local Search heuristic improves upon the ideas of Peacock and seems to approximate more precisely. While Peacock does not approximate as good as Local Search, it is much faster and should be preferred if speed is a priority. The Short Path heuristic is interesting in theory, and we consider it worth of further investigations, but currently it behaves worse than the other approaches.

Like for Chapter 3, our code is publicly available at `https://github.com/RaduVintan/sdn/` and can be used to test all algorithms and reproduce all results presented in this chapter.

4.1 The Peacock Algorithm

4.1.1 Preliminary Framework

The Peacock algorithm, as already mentioned, has been introduced in [6, 7]. In this section we construct a suitable novel framework for explaining both Peacock and our new algorithms. It partially uses notions already introduced in [7], such as node merging, but we also introduce new concepts to later ensure a unified approach.

Definition 14 (Node merging)**.** Let G be the *full graph* of some instance (n, σ) of the RLF Problem. For any node $i \in \{1, \ldots, n-1\}$, let $out(i) := \sigma(\sigma^{-1}(i) + 1)$ be the successor of i w.r.t. the new path. When we update i, we implicitly *merge* i with $out(i)$ in the full graph. This amounts to changing G in the following way:

- Node i is removed and any edges pointing into i will instead point into $out(i)$.
- The old edge $(i, i+1)$ and the new edge $(i, \sigma(i))$ are removed.

i.e. we treat i and $out(i)$ as a single node, with incoming rules from both i and $out(i)$, but outgoing rules just from $out(i)$ [7]. Intuitively, merging i with $out(i)$ models the fact that, once updated, i will only delegate any incoming packets to its new successor $out(i)$.

Definition 15 (Update Tree)**.** Let $S \subseteq \{1, \ldots, n\}$ be a subset of already updated nodes for an instance (n, σ) of the RLF Problem with *full graph* G. We represent this in the full graph by merging all nodes $i \in S$ with their corresponding successors $out(i)$. Note that if both i and $out(i)$ are merged, then they will both be merged into $out(out(i))$. It is easy to observe that, independently of the order in which the nodes from S are merged, the same *update tree* $G(S)$ is obtained. $G(S)$ is a tree with regards to the remaining old edges (hence the name), while the remaining new edges form a permutation over its nodes. (This can be trivially proven using induction over the number of merge operations.)

We can visualize the concepts introduced by the previous two Definitions in Figure 4.1. In the first round, nodes 2 and d are merged. In the second round, node 3 is merged to node 5, which is in turn merged with node 4, which is in turn merged with node $2, d$. This leads to the update tree on the right, which is in fact the full graph corresponding to the only instance of the RLF Problem with $n = 2$ nodes. Other examples of node merging are available in [7].

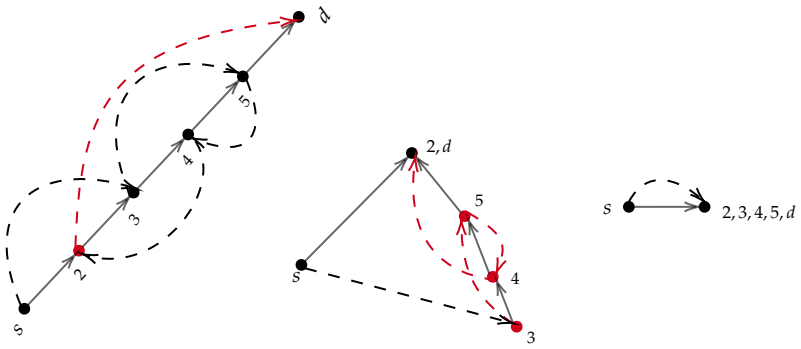


Figure 4.1: Running Peacock on an instance with 6 nodes. From left to right, we see the update trees $G(\emptyset), G(\{2\})$ and $G(\{2,3,4,5,d\})$. Next to each node we also write out the indexes of the nodes which have been merged into it.

The following Lemma contains what we consider to be the unifying idea behind Peacock and also our new algorithms from Sections 4.2.2 and 4.2.3.

Lemma 7 (Main Lemma)**.** *Let G be the full graph of an instance (n, σ) of the RLF Problem. Let P be a path in G from the source 1 to the destination n. Let P_{old} and P_{new} be the nodes in P which use their old and respectively their new outgoing edge to form the path P. Let $S := \{1, \ldots, n-1\} \setminus P_{old}$ and consider the update tree $G(S)$. Then $G(S)$ is the full graph of an instance (k, τ) of the RLF Problem, where $k := |P_{old}| + 1$ and τ is a permutation over the nodes in $P_{old} \cup \{n\}$*

Proof. Updating all nodes in $S := \{1, \ldots, n-1\} \setminus P_{\text{old}}$, i.e. merging all these nodes i with their corresponding successors $out(i)$ in G, produces a graph $G(S)$ defined on the non-merged nodes, which are given by $V' := P_{\text{old}} \cup \{n\}$. In general, as mentioned in Definition 15, $G(S)$ is a tree w.r.t. the remaining old edges of the full graph, and the remaining new edges form a permutation τ over the remaining nodes V'. To prove that $G(S)$ is the *full graph* of another instance of the RLF Problem, it therefore suffices to prove that the remaining old edges, call them E', fulfill the following property:

- The edges in E' form a directed path over all the nodes in V'. The first node s in this path has no ingoing edges (old or new), the last node d in this path has no outgoing edges (old or new).

By definition, no merge operation can change the number of outgoing old edges of a remaining node. Hence, all nodes in V', except n, have exactly one outgoing old edge. Now

assume that some node $i \in V'$ has (at least) two ingoing old edges: (j_1, i) and (j_2, i) with $j_1 \neq j_2$ and $j_1, j_2 \in V'$. As j_1 and j_2 have outgoing old edges, we have $j_1, j_2 \neq n$, and hence $j_1, j_2 \in P_{\text{old}}$. Let $j_1' := j_1 + 1 \in P$, $j_2' := j_2 + 1 \in P$. We have $j_1' \neq j_2'$. As $(j_1, i), (j_2, i) \in E'$, it follows that j_1' and j_2' have both been eventually merged into the node i, possibly by a sequence of ≥ 0 merge operations. Merges are per definition only possible along new edges in the full graph G. Hence, there are subpaths of P consisting exclusively of new edges (in G) from j_1' and j_2' to i. But, because j_1', j_2' are distinct and included in P, this implies that some node has two ingoing new edges in the path P, contradiction. It follows that all nodes in V' have at most one ingoing edge. Since $G(S) := (V', E')$ is a tree, it follows that the edges in E' form a path over the nodes in V'. Hence, there exists a node s with no ingoing old edges. (Recall that n is the node with no outgoing old edges.) Because merge operations cannot change the number of outgoing new edges of a remaining node, it also follows that n has no outgoing new edges.

s has no ingoing old edge, so either $s = 1$ or the node 1 has been eventually merged into s (possibly by a sequence of multiple merge operations). In the former case, it trivially follows that $s = 1$ has no ingoing new edges, for the reason that merge operations cannot change the number of ingoing new edges of a node. In the latter case, if s has an ingoing new edge, this would imply that s had at least two distinct ingoing new edges in G, because one of them was required to merge 1 into s, which is a contradiction. □

The Main Lemma offers us a template to design algorithms for the RLF Problem in the following way:

Algorithm 5: Template for RLF Algorithms

```
1 Input: instance of RLF given by full graph G with n nodes.
2 if n = 1 then
3 |   return 1
4 Fix a path P from 1 to n in the full graph G.
5 Update the nodes in S := {1, ..., n − 1} \ P_old using some amount of rounds r.
6 Solve the remaining instance given by G(S) recursively in r' rounds.
7 return r + r'
```

One needs to be careful when using the template given in Algorithm 5. If for example we choose P to be the path given by the new edges E_{new}, then $P_{\text{old}} = \emptyset$ and Step 5 is equivalent to the initial problem. We therefore need to choose a path such that P_{old} is not empty. On the other hand, if we choose the path P given by the old edges E_{old}, then $S = \emptyset$ and we update nothing (also $G(S) = G$). We therefore need to choose a path such that P_{old} is not too large. As we will see, both Peacock and our algorithms from Sections 4.2.2 and 4.2.3 will use different versions of this general idea.

4.1.2 Basics

We are now ready to present the idea behind the Peacock algorithm using our framework from the previous section. Recall the concepts of *forward* and *backward edge/node*, first introduced in Section 3.1: In the corresponding *full graph* G of an instance with n nodes and permutation σ of the RLF Problem, a *new* edge $e := (i, j) \in E_{\text{new}}$ is called *forward* iff $j > i$, i.e. if it jumps ahead w.r.t. the old path. Otherwise e is called *backward*. The node i is called *forward* or *backward node* respectively. We provide some further definitions:

Definition 16 (Forward Path)**.** A path P from 1 to n in the *full graph* G of an instance of the RLF Problem is called *forward path* if it does not contain any backward edges. Then, P is made up exclusively of old edges and of (new) forward edges.

Definition 17 (Valid Forward Edge Choice)**.** Identify each forward edge $e := (i, j)$ (with $i < j$) of a *full graph* G with the interval $[i, j]$. Let $Forw$ be the set of forward edges of G. Consider the *set of valid forward edge choices* given by:

$$\begin{aligned}\mathcal{F} := \{F \subseteq Forw :\ &\text{intervals corresponding to forward edges in } F \\ &\text{share no common nodes except potentially the endpoints}\}\end{aligned}$$

We call $F \in \mathcal{F}$ a *valid choice (of forward edges)*. If there exists no $F' \in \mathcal{F}$ such that $F \subset F'$ we call F a *maximal valid choice (of forward edges)*

Notice that there exists a *forward path* P_F containing exactly the forward edges of some subset of forward edges $F \in forw$ if and only if we have $F \in \mathcal{F}$.

Both for the SLF and the RLF Problems, it is trivial to observe that one can update any forward nodes in the first round. This cannot create any cycles. However, for the RLF Problem in particular, a second observation can be made: If a forward node i with corresponding new

edge (i, j), $j > i$ is updated in the first round, then all nodes $i < k < j$ can be safely updated in the next/second round, since they are unreachable from the source (and hence any cycles are non-problematic). Taking this thought even further leads us to the following:

Lemma 8. *Let P be a forward path in the full graph G of an RLF Problem instance. Then one can reduce this instance to the smaller instance $G(S)$, where $S := \{1, \ldots, n-1\} \setminus P_{old}$, in just two rounds. In the first of these rounds we update the new forward edges in P, and in the second round we can update all other nodes from S.*

In [7], a forward path P_F is constructed starting from a *maximal valid choice of forward edges* F in the following way: first, one sorts all forward edges in descending order of their length; then, one greedily adds forward edges to the current subset F of forward edges, starting from the longest one, while keeping the invariant that F is *valid* (and hence P_F is well defined per Definition 17). It is then shown that:

Lemma 9 ([7])**.** *The above strategy for constructing the forward path $P := P_F$ ensures that the reduced instance $G(S)$, where $S := \{1, \ldots, n-1\} \setminus P_{old}$ has at most $2n/3$ nodes, where n is the size of the initial instance given by G.*

Combining Lemmas 8 and 9 then allows to conclude that Peacock requires only $\mathcal{O}(\log n)$ rounds for any RLF Problem instance. An example of Peacock running is shown in Figure 4.1, which we also used to understand how node merging works. The forward path P is given by $(1, 2, 6)$ and thus $S := \{2, 3, 4, 5\}$. We update 2 in the first round and $3, 4, 5$ in the second round, as argued by Lemma 8. This leads to the new RLF Problem instance given by $G(S)$, which can be solved one round, leading to a total of 3 rounds.

4.1.3 Graphs requiring $\mathcal{O}(\log n)$ rounds

In [7] a family G_j of graphs is presented (instances of the RLF Problem), each having about 2^j nodes, for which Peacock requires about $j = \Omega(\log 2^j)$ rounds to schedule. This shows that the bound $\mathcal{O}(\log n)$ for the number of rounds Peacock requires is tight. In this section we point out some small mistakes in [7] regarding the definition of the graphs G_j. We then proceed to present a corrected version which we also consider easier to understand than the version in the paper.

Identifying problem in construction

The graph G_j has $N_j := 8 \cdot 2^j$ nodes. The idea of the construction from [7] is to define G_j such that applying Peacock for two rounds on G_j produces G_{j-1}. This then easily allows to prove that G_j is solved by Peacock in about $j = \Omega(\log N_j)$ rounds. For each $3 \leq i \leq 2^j + 2$, we have the 8 nodes:

$$v_{1/8-\frac{1}{2^i}}, v_{2/8-\frac{1}{2^i}}, \ldots, v_{8/8-\frac{1}{2^i}}$$

The initial path in G_j goes through the nodes in ascending order of their index. To simplify the notation, we let $v_{ki} := v_{k/8-\frac{1}{2^{i+2}}}$ for any $k \in \{1, \ldots, 8\}$ and $1 \leq i \leq 2^j$. With this change of notation, the old path can be written as:

$$s := v_{1,1} \to v_{1,2} \to \cdots \to v_{1,2^j} \to \cdots \to v_{8,1} \to v_{8,2} \to \cdots \to v_{8,2^j} = d \tag{4.1}$$

The new path is determined by the new edges, whose construction we describe. The forward edges go from the first half to the second half of the old path in the following way:

$$v_{1,1} \to v_{5,1},\ v_{1,2} \to v_{5,2}, \ldots,\ v_{4,2^j-1} \to v_{8,2^j-1},\ v_{4,2^j} \to v_{8,2^j} \tag{4.2}$$

i.e. from each node in the first half the path jumps over half of the nodes into the second half.

The other new edges, all backward and starting from the nodes of the second half, are defined by specific rules. We write the rules for the nodes with $k \in \{5, 6\}$ and the others separately for reasons that will become clear later. For $k \in \{5, 6\}$, the backward edges are defined as:

$$v_{ki} \to v_{k-2,i} \tag{4.3}$$

For the others:

$$v_{7i} \to v_{2i}, v_{8i} \to v_{1,i+1} \tag{4.4}$$

where for the last rule we exclude $i = 2^j$ because the destination $d = v_{8,2^j}$ has no outgoing edges.

Unfortunately, the rules at (4.4) are problematic, as we now show. Consider G_2, having 32 nodes. Applying the first two rounds of Peacock to this graph should produce G_1, having 16 nodes. As discussed in [7], Peacock updates the edge $v_{1,1} \to v_{5,1}$ in the first round, and then all nodes from the first half (i.e. those with $k \leq 4$) in the second round. This means that each node v_{ki} with $k \leq 4$ will be merged with its sibling $v_{k+4,i}$. Call this graph G_2''. Node 13 (per the ordering of the old path) of G_2'' is $v_{8,1}$. In G_2 the backward path from $v_{8,1}$ led to $v_{1,2}$ (per (4.4)), which was then merged with $v_{5,2}$, which is Node 2 in G_2''. We conclude that in G_2''

we have a (new) backward edge from Node 13 to Node 2. However, in G_1, which should be identical to G_2'', this does not hold. Node 13 in G_1 is $v_{7,3}$, whose new edge points to (per rule (4.4)) $v_{2,3}$, which is Node 3 in this graph.

This problem has been confirmed (and discovered) by implementing the graphs G_j as described by the above rules. Furthermore, scheduling any of these graphs (for any j) with Peacock can be done in at most 7 rounds, which confirms that the error breaks the main result regarding these graphs. In the following we will show that we can keep the rules (4.1), (4.2), (4.3), while only adjusting the rules from (4.4). We do this by defining the graphs G_j somewhat differently. We then prove that Peacock indeed needs $\Omega(\log n)$ rounds for these graphs, and also confirm this fact experimentally.

Fixing the construction

First, we drop the convention that G_j has $8 \cdot 2^j$ nodes and instead assume from now on that G_j is the graph with 2^j nodes. We start with G_1, which is the instance with two nodes where the new edge is parallel to the old edge, and define G_j for $j \geq 2$ by recursively using G_{j-1}. Note that to define G_1 the assumption from previous sections that the sets of new and old edges are disjoint is temporarily dropped. Assume $j \geq 2$ and that G_{j-1} has already been constructed. We keep the invariant that G_j has forward (new) edges from the nodes of the first half into the second half, and backward (new) edges from the nodes of the second half into the first half.

The graph G_j has nodes $1, \ldots, N_j$ where $N_j := 2^j$. To ease the notation let $N := N_j$. The rules for the first $3N/4$ nodes are identical to the rules (4.2) and (4.3) we used previously. More precisely, we have the new edges:

$$i \to i + N/2, \forall i \in \{1, \ldots, N/2\} \text{ and } i \to i - N/4, \forall i \in \{N/2 + 1, \ldots, 3N/4\} \tag{4.5}$$

Hence, the remaining backward edges are required to go from the nodes $3N/4 + 1, \ldots, N - 1$ into the nodes $2, \ldots, N/4$. To define the rules for these nodes we consider the graph G_{j-1}, which by the invariant has $N/2$ nodes and $N/4 - 1$ backward edges (i.e. the same number of backward edges which are yet to be defined for G_j). For $i \in \{N/4 + 1, \ldots, N/2 - 1\}$, let $f(i) \in \{2, \ldots, N/4 - 1\}$ be the node to which the backward edge from i points to in G_{j-1}. Note that (again by the invariant) f is a bijection from $\{N/4 + 1, \ldots, N/2 - 1\}$ to $\{2, \ldots, N/4 - 1\}$. We use the same bijection f to map in G_j the nodes from $3N/4 + 1, \ldots, N - 1$ into the nodes

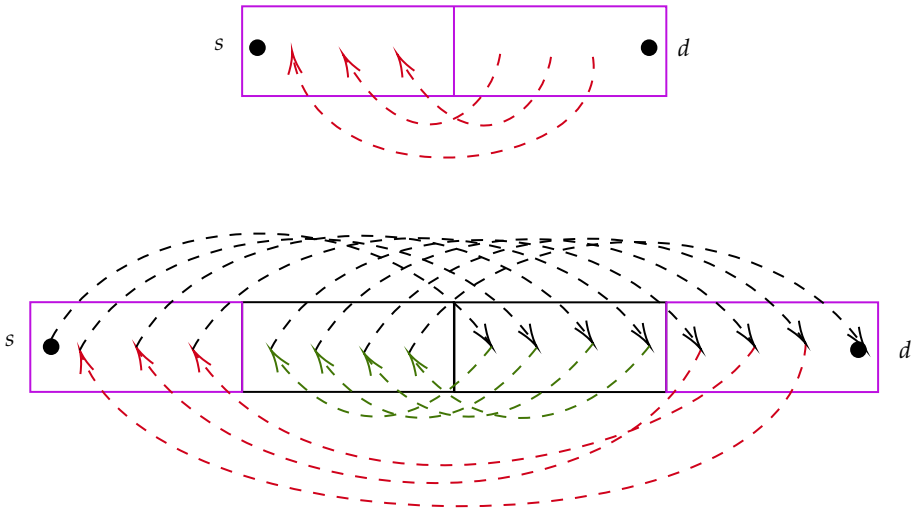


Figure 4.2: Showing how to obtain G_4 from G_3. The backward edges from the second half to the first half in G_3 become backward edges from the last quarter to the first quarter in G_4.

$2, \ldots, N/4$. Hence, our last rule is:

$$i \to f(i - N/2), \forall i \in \{3N/4 + 1, \ldots, N - 1\} \tag{4.6}$$

and it is easy to check that G_j also fulfills the invariant. Figure 4.2 shows this recursive construction.

To prove that two steps of Peacock on G_j lead to G_{j-1} (for $j \geq 2$), we recall that these steps are equivalent to merging all nodes from the first half into their corresponding siblings from the second half, i.e. we merge each i with $i + N/2$ for $i \in \{1, \ldots, N/2\}$. (We recommend looking at Figure 4.2 again to understand the following explanations.) We obtain a graph G_j'' with $N/2$ nodes. The first $N/4$ nodes of this graph had (in G_j) new edges pointing into the second quarter of G_j, which was then merged with the fourth/last quarter. Hence, in G_j'', we get the right behavior that the nodes from its first half have forward (new) edges into the nodes from the second half.

As for the backward edges in G_j'', by construction of G_j they mimic the behavior of the backward edges from G_{j-1}. Let i be a node from the last quarter of G_j (and hence second half of G_j''). Its backward edge points to $f(i - N/2)$ in G_j, and this node was merged with

$f(i - N/2) + N/2$, which is the $f(i - N/2)$-th node in G_j''. But node i in G_j is node $i - N/2$ in G_j'', which shows that each node v from the second half of G_j'' has a backward edge to $f(v)$ in G_j'', which is the same behavior as in G_{j-1}. The proof is complete.

A second observation

In [7], the authors claim that the optimal solution for their graphs G_j uses 4 rounds for the graphs which have 16 to 256 rounds, and 7 rounds for the graph with 512 nodes. First we remark that this contradicts the idea that two rounds of Peacock on G_j lead to G_{j-1}. Indeed: if this were true, a valid strategy for the 512-node graph would be to apply two steps of Peacock to reach the graph with 256 nodes and then solve the remaining instance in 4 rounds, totaling $6 < 7$ rounds (which is the claimed optimum). This seems therefore to confirm that the graphs G_j from [7] fail to have the required recursive property. Moreover, we implemented the graphs G_j from the paper and could confirm that for instances having 16, 32 or 64 nodes the optimum is indeed 4.

We also implemented the "correct" graphs G_j (recall that in our notation they each have 2^j nodes). Applying Peacock to G_j experimentally confirms that this requires $2j - 1 = \mathcal{O}(\log n)$ rounds. We also computed the optimum for these graphs using our ILP formulation, and obtained for $j \in \{1, \ldots, 6\}$ that G_j can be scheduled optimally in j rounds, which contradicts the idea that instances with 16, 32 or 64 nodes are equally hard.

4.1.4 Approximation Factor Analysis

We prove the following:

Theorem 2. *For infinitely many n, there are instances of the RLF Problem for which the Peacock algorithm requires $\Omega(\log n)$ rounds, but which can be solved optimally using only constantly many rounds. Hence, Peacock is not an $o(\log n)$ approximation.*

Proof sketch (also see Figures below). Consider the (corrected) graphs G_j from the previous section. These graphs correspond to instances of the RLF Problem, and have $n = 2^j$ nodes each. The Peacock algorithm requires $\Omega(\log n)$ rounds to schedule these instances. Also, running the first two rounds of Peacock on G_j produces G_{j-1}. Now fix some j (and thus n) and construct the instance given by the graph G as shown in the upper part of Figure 4.3 and described in the following: In the initial path, the source node s is followed by the nodes of

the graph G_j, then by a node v, then by the nodes of another graph D_j (corresponding to another/any instance of the RLF Problem), and then by a node w and the destination node d.

The graphs D_j and G_j both have n nodes. Let (s_1, d_1) and (s_2, d_2) be the source-destination pairs of the graphs G_j and D_j respectively. To make the graph G correspond to an instance of the RLF Problem we still need to completely specify the new path. The new path starts at s, then goes to v, then to s_1 and through the new edges in G_j eventually to d_1, then to w, to s_2 and through the new edges in D_j eventually to d_2, and then finally to d.

Figure 4.3 explains the first two rounds of Peacock for the graph G. It shows the corresponding *update trees*. In the first round, the longest forward edge is (d_1, w) and therefore d_1 is updated (hence merged with w), leading to the configuration shown in the middle of the Figure. Both G_j and D_j were replaced by G'_j and D'_j: these are the *update trees* G_j and D_j on which the first round of Peacock was applied. In the second round, v and all nodes inside D'_j are going to be updated, leading to the configuration shown in the bottom part of the Figure. Notice how the graph G'_j was replaced by G_{j-1}. But now it is easy to see that Peacock will still require $j - 1 = \log n$ rounds to finish solving this instance.

To solve the instance given by G faster, we update the edge node s in the first round (instead of d_1). Then, in the second round we are able to update all nodes inside G_j and reach the graph shown in Figure 4.4. If we now run Peacock on this instance it is easy to see that it will lead to a schedule with constantly many rounds (5 in total). Since j (and thus n) were chosen arbitrarily big this settles the proof. □

4.2 Other Algorithms

4.2.1 LP Relaxation

Our LP algorithm for the RLF Problem closely resembles the algorithm from Section 3.3.2 for the SLF Problem. In fact, we only need to adapt the cycle constraints of the polytope $LP(T)$. Recall that for the relaxed problem one is allowed to have cycles in the *update graph* as long as they are not reachable from the source node s. To write these constraints properly, we use the following:

Definition 18. Let (n, σ) be an instance of the RLF Problem and $G := (V, E)$ the *full graph*

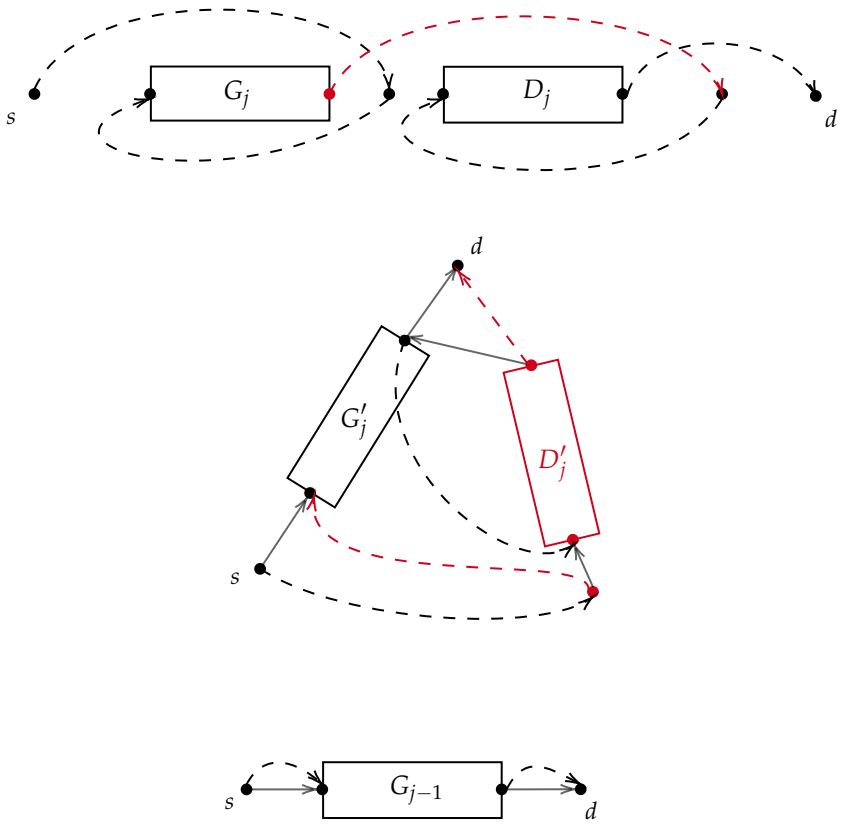


Figure 4.3: Running first two rounds of Peacock on our instance leads to a situation where $j - 1 = \log n$ rounds will be required. Grey full edges correspond to the old path; black or red dashed edges correspond to the new path. Red elements are updated in the following shown round.

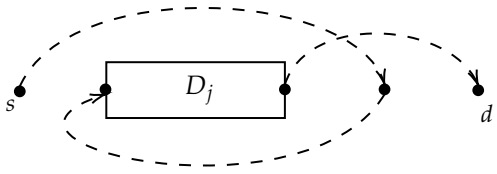


Figure 4.4: Configuration reached by a RLF solution which updated s in the first round and then all nodes inside G_j in the second round.

associated to this instance. Let $\mathcal{S}$ be a set such that:

$$S := (P, v) \in \mathcal{S}$$

iff $P := (v_1 := s, v_2, \ldots, v_k)$ is a tuple of $k \geq 1$ distinct nodes, such that there exists the path P in G, $(v_k, v) \in E$ and $v = v_i$ for some $1 \leq i \leq k$. This means that the *S-tuple* $(v_1 = s, \ldots, v_k, v)$ forms a path from s to some node v_k, together with an edge from v_k to an already visited v node closing a cycle (reachable from s). We call such a sequence of edges a *path-cycle structure*.

For $S := (P, v) \in \mathcal{S}$ and corresponding *S-tuple* $(v_1 = s, \ldots, v_k, v)$, let S_{old} and S_{new} be the vertices from $\{v_1, \ldots, v_k\}$ which use their old and respectively their new outgoing edge to form the corresponding *path-cycle structure*. Similarly to how the old and new edges partition E (i.e. an edge is either old or new), S_{old} and S_{new} partition the vertices $\{v_1, \ldots, v_k\}$ appearing in the *path-cycle structure*.

Lemma 10. *Let $T \geq 2$ be a natural number. We define the variables y_{ti} for all $t \in \{0, \ldots, T\}$ and $i \in [n]$. Consider the polytope $LP(T)$ given by the following set of constraints:*

$$\begin{aligned}
y_{0i} &= 1, \quad \forall i \in [n] \\
y_{Ti} &= 0, \quad \forall i \in [n] \\
x_{ti} + y_{t-1,i} &= 1, \quad \forall t \in [T], i \in [n] \\
y_{t-1,i} - y_{ti} &\geq 0, \quad \forall t \in [T], i \in [n] \\
\sum_{i \in S_{old}} x_{ti} + \sum_{y \in S_{new}} y_{ti} &\geq 1, \quad \forall t \in [T], \forall S \in \mathcal{S}
\end{aligned}$$

Then, the constraints can be satisfied by integral variables iff there is a schedule with T rounds for the corresponding instance (n, σ) of the RLF problem.

Proof. The proof is almost identical to the proof of Lemma 5 and left to the reader. It is easy to check that the new cycle breaking constraints model the RLF property. □

To find a feasible fractional solution (or show that no such solution exists) in polynomial time, we can use the Ellipsoid method with a suitable separation oracle, which we can again implement using the Floyd-Warshall algorithm. Unlike in the SLF case, where we used Floyd-Warshall to find the shortest cycle, here it is used to find the shortest *path-cycle structure*. To do that, it suffices to compute for all nodes i the length l_i of the shortest cycle C_i containing i, and the length l'_i of the shortest path from the source s to the node i (with regards to the distances given by x_{ti} for old edges and y_{ti} for new edges). If, for all nodes i, it holds that

$l_i + l'_i \geq 1$, then the cycle breaking constraints are fulfilled. Otherwise, if for some i we have $l_i + l'_i < 1$, then we can easily extract the set $S \in \mathcal{S}$ which violates its corresponding constraint.

Naturally, we can also use the polytope to compute the optimal *integral* solution OPT, i.e. the optimal schedule for the corresponding RLF instance, albeit not in polynomial-time. To do that, we just have to search for *integral* feasible points inside $LP(T)$ instead of searching for *fractional* feasible points. (The similar discussion from the end of Section 3.3.3 applies.)

Implementation details

Like discussed at the end of Section 3.3.3, for reasons of efficiency we initially add no cycle constraints at all, and search for (either fractional or integral, depending on what is computed) feasible points inside this relaxed version of $LP(T)$. If no feasible point is found, T needs to be increased. Otherwise, we check with Floyd-Warshall if any cycle constraint is violated. If not, then we are done. Otherwise, we add the violated constraints corresponding to all nodes i for which $l_i + l'_1 < 1$ in Gurobi and solve the new instance. This lazy addition of constraints turns out to lead to much better performance than an implementation which includes all (possibly unnecessary) cycle constraints at the start.

4.2.2 Local Search

In Section 4.1.1 we presented Algorithm 5, which consists of a template for designing further algorithms for the RLF Problem. Then, in Section 4.1.2 we saw that one can ensure fast updates by choosing a *forward path* P (see Definition 16) and using Lemma 7 to obtain smaller instances in just two rounds. The Peacock algorithm chooses the forward path P_F starting from a greedily determined *maximal valid choice of forward edges* F (see Definition 17).

Our idea for an improved algorithm is to choose the set F more carefully. We will also use a *maximal valid* such subset F, as per Definition 17, and then use the induced *forward path* P_F to reach smaller subinstances. We determine the subset F using Local Search as shown in Algorithm 6.

There are some details that require further attention. Firstly, at Line 3, the initial choice of a set F can be made in multiple ways. One way would be to start with the solution F chosen by Peacock. This guarantees that Local Search is at least as good of an approximation as Peacock itself. For our implementation, however, we wanted to avoid making Local Search artificially more accurate than Peacock, and therefore we start with a somewhat arbitrary F: we go over

Algorithm 6: Local Search with 1-Neighborhood

```
1  if n = 1 then
2  |  return 1
3  F ∈ 𝓕 some initial maximal valid choice of forward edges
4  P := P_F and corresponding S (recall S := {1, ..., n − 1} \ P_old)
5  r(S) → LocalSearch(G(S)) (recursive call)
6  while true do
7  |  for f ∉ F do
8  |  |  F' ← {f}
9  |  |  add all edges from F to F' that can be added without making F' invalid
10 |  |  add other arbitrary forward edges to F' if it is not yet maximally valid
11 |  |  P' := P_F' with corresponding S'
12 |  |  if LocalSearch(G(S')) < r(S) then
13 |  |  |  improve solution: change F to F', P to P_F', S to S'
14 |  end
15 |  if no improvement found then
16 |  |  break loop
17 end
18 return 2 + LocalSearch(G(S)) (the 2 comes from updating S)
```

the forward edges from left to right and add them to F as long as this does not break validity (as defined in Definition 17). At Line 9, the forward edges that are to be added to ensure maximality of F' can again be chosen in any arbitrary way. For our implementation, we again go left to right and add a forward edge to F' iff this does not break validity.

We call a solution of Local Search *strictly local* if there exists another (valid) choice F of forward edges which would have led to a better solution than the one obtained by the algorithm. This can happen if the algorithm fails to explore this particular F during its search at Lines 7 to 14. Notice that, if different initial choices of F at Line 3 lead to solutions with different number round rounds (for the same instance), then the worse of these solutions is certainly a *strictly local* solution. When searching for neighboring solutions, Algorithm 6 uses

a 1-neighborhood: at Line 9, it tries changing the current solution by initially including *one* forward edge which is not present in the current solution. One could imagine a generalization of our approach where at Line 9 we initialize F' using $k \geq 1$ currently unused forward edges (assuming they do not break validity of F'). However, our experiments indicate that there will always exist instances for which a *strictly local* solution is reached, independently of the size of the neighborhood, and we therefore decided to limit the routine to 1-neighborhoods for reasons of efficiency. As we will see in Section 4.2.4 this is enough to make the Local Search algorithm the best performing heuristic.

A last remark: recall the hard instance discussed in Section 4.1.4 and shown in Figure 4.3, which Peacock solves in $\mathcal{O}(\log n)$ rounds even though it can be solved in $\mathcal{O}(1)$ rounds. It is easy to see that the Local Search algorithm also finds a schedule with $\mathcal{O}(1)$ rounds for this instance, as it tries out (among different possibilities) updating s in the first round. This choice leads to a very short solution, as already discussed at Figure 4.4. In fact, unlike in the case of Peacock, we could not prove that the Local Search algorithm is not an $o(\log n)$-approximation.

4.2.3 Short Path

Peacock and Local Search both use *forward paths* and apply the (Main) Lemma 7 to construct fast schedules. The advantage of using *forward paths* P is the fact that the reduced instance $G(S)$ can be reached in only two rounds, per Lemma 8. We begin by generalizing this Lemma to arbitrary paths:

Lemma 11. *Let P be a path from 1 to n in the full graph G of an RLF Problem instance, and $S := \{1, \ldots, n-1\} \setminus P_{old}$. Notice that the set P_{new} can be partitioned into the set of forward and backward nodes (which use their new forward or backward in the path P). Call these sets P_{forw} and P_{backw} respectively. The sets P_{old}, P_{forw} and P_{backw} partition the nodes in the path P (except n).*

One can then reach the subinstance $G(S)$ in just $2 + |P_{backw}|$ rounds.

Proof. In the first round we update the nodes in P_{forw}, in the rounds 2 to $|P_{\text{backw}}| + 1$ we update the nodes in P_{backw} starting from left to right, and in the last round we update all other nodes from S. This is a valid schedule for the RLF Problem. □

Lemma 11 tells us that $|P_{\text{backw}}|$ additional rounds are required for using a path P containing backward edges in comparison to using a *forward path*. Why would we use such a path? Well, recall that the subinstance $G(S)$ contains precisely $1 + |P_{\text{old}}|$ nodes, and hence it can be solved

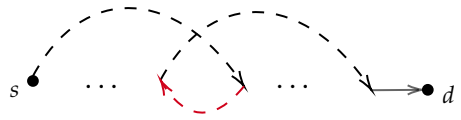


Figure 4.5: One type of instance for which there is a path P from s to d containing one backward edge, having the advantage that $|P_{\text{old}}| = 1$. For such an instance the Short Path Heuristic finds a solution in $\mathcal{O}(1)$ rounds.

in at most $|P_{\text{old}}|$ rounds. (Of course, even in $\mathcal{O}(\log |P_{\text{old}}|)$ many rounds, as we know from the introduction to Peacock.) However, it can happen that some paths P for which $|P_{\text{old}}|$ is very small contain backward edges, as exemplified in Figure 4.5. Therefore, sacrificing $|P_{\text{backw}}|$ additional rounds to update those backward edges can be a good compromise if $|P_{\text{old}}|$ is in turn very small. The following Corollary summarizes these thoughts:

Corollary 12. *Let P be a path from 1 to n in the full graph G of an RLF Problem instance. The instance can be scheduled in at most:*

$$2 + |P_{old}| + |P_{backw}|$$

rounds. It is therefore a viable strategy to search for paths P for which the quantity $|P_{old}| + |P_{backw}|$ is as small as possible.

We obtain the following algorithm, which we call Short Path:

Algorithm 7: Short Path

1 **if** $n = 1$ **then**
2 | return 1
3 Compute 1-n path P with minimal $|P_{\text{old}}| + |P_{\text{backw}}|$ in full graph G using the Dijkstra algorithm (or any other single source shortest path algorithm).
4 $S := \{1, \ldots, n-1\} \setminus P_{\text{old}}$
5 Compute reduced instance $G(S)$.
6 $r(S) \leftarrow$ ShortPath($G(S)$) (recursive call)
7 return 2 + $|P_{\text{backw}}|$ + $r(S)$

4.2.4 Experiments

As in Section 3.3.4, we gathered statistical data to understand the behavior of our algorithms in practice and to compare their performance. Figure 4.7 uses Boxplots to show the distribution of LP_{OPT}, OPT and the results obtained by the LP Rounding Algorithm, the Peacock Algorithm, the Local Search Algorithm and the Short Path Algorithm for randomly generated permutations/instances of different sizes for the RLF Problem. The results are very similar throughout the four shown Boxplots, which suggests that the number of nodes does not play a significant role in the relative performance of the algorithms.

As for the SLF Problem, the quantities LP_{OPT} and OPT have a narrow distribution and are almost always equal to each other; for most instances only 3 or 4 rounds are required. The LP Rounding Algorithm has a broad distribution with long upper whiskers and outliers. A similar pattern can be observed for the Short Path Algorithm. Its box lies higher and suggests that this algorithms performs the worst when it comes to approximation quality. Peacock solves almost all instances in 5 rounds, but outliers exist at 3 and 7 rounds respectively. In all cases, the Local Search Algorithm seems to perform the best: all its results fit compactly in a small box which lies almost as low as the boxes of LP_{OPT} and OPT. There are no whiskers or outliers. In Table 4.1 we gather some information regarding the empirical approximation factors. For space reasons we only show the results for $n = 70$ and $n = 85$. They are already suggestive and confirm that Local Search is the best performing approximation. While Short Path and Peacock are less precise, they are also faster.

In Figure 4.6 we compare the non-LP based heuristics for larger values of n. (The LP cannot be used for such big instances.) For each value of $n = 300, 310, \ldots, 400$ we sample 50 random permutations and compute the mean number of rounds required to solve these instances for all algorithms. The results confirm the impressions from the previous paragraph: Local Search finds the shortest schedules, Short Path finds the longest schedules, and Peacock lies in-between.

Comparison with another solver

Like in Section 3.3.4, we again compare our optimal ILP solver with the ILP solver implemented in [7] (and adapted from an ILP in [14]). Again, we could verify that the two implementations offer the same results and are therefore likely both correct. For example,

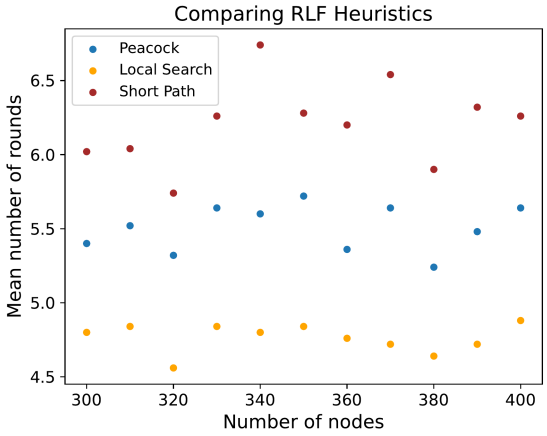


Figure 4.6: RLF: Comparison of heuristics for large values of n

	Maximum difference	Maximum factor	Mean difference	Mean factor
Rounding ($n = 70$)	7	2.75	1.29	1.35
Peacock ($n = 70$)	4	2.33	1.26	1.37
Local Search ($n = 70$)	2	1.66	0.74	1.2
ShortPath ($n = 70$)	4	2.33	1.52	1.44
Rounding ($n = 85$)	5	2.25	0.93	1.24
Peacock ($n = 85$)	4	2.33	1.36	1.41
Local Search ($n = 85$)	2	1.66	0.66	1.18
ShortPath ($n = 85$)	4	2.33	1.58	1.46

Table 4.1: RLF: Empirical approximation factors.

both ILPs confirm that the optimal solution for the graphs G_j from [7] (presented in Section 4.1.3) is indeed 3 rounds for networks of size 8 and 4 rounds for networks of size 16, 32 or 64. This confirms the claim in [7]. The corrected versions of the graphs G_j indeed requires 3 rounds for networks of size 8, 4 rounds for for size 16, 5 rounds for size 32 and 6 rounds for size 64. This is a further confirmation of Theorem 2 and of our comments from Section 4.1.3.

Regarding speed, the situation is similar to the one observed in Section 3.3.4. Our program for RLF runs all experiments required for computing OPT for Figure 4.7a (70 nodes and 150 permutations) in about 20 minutes, while the other ILP solver requires over 15 minutes for just the first 15 generated permutations. Hence, we conclude that our algorithm is the fastest in practice we know for optimally solving the RLF Problem.

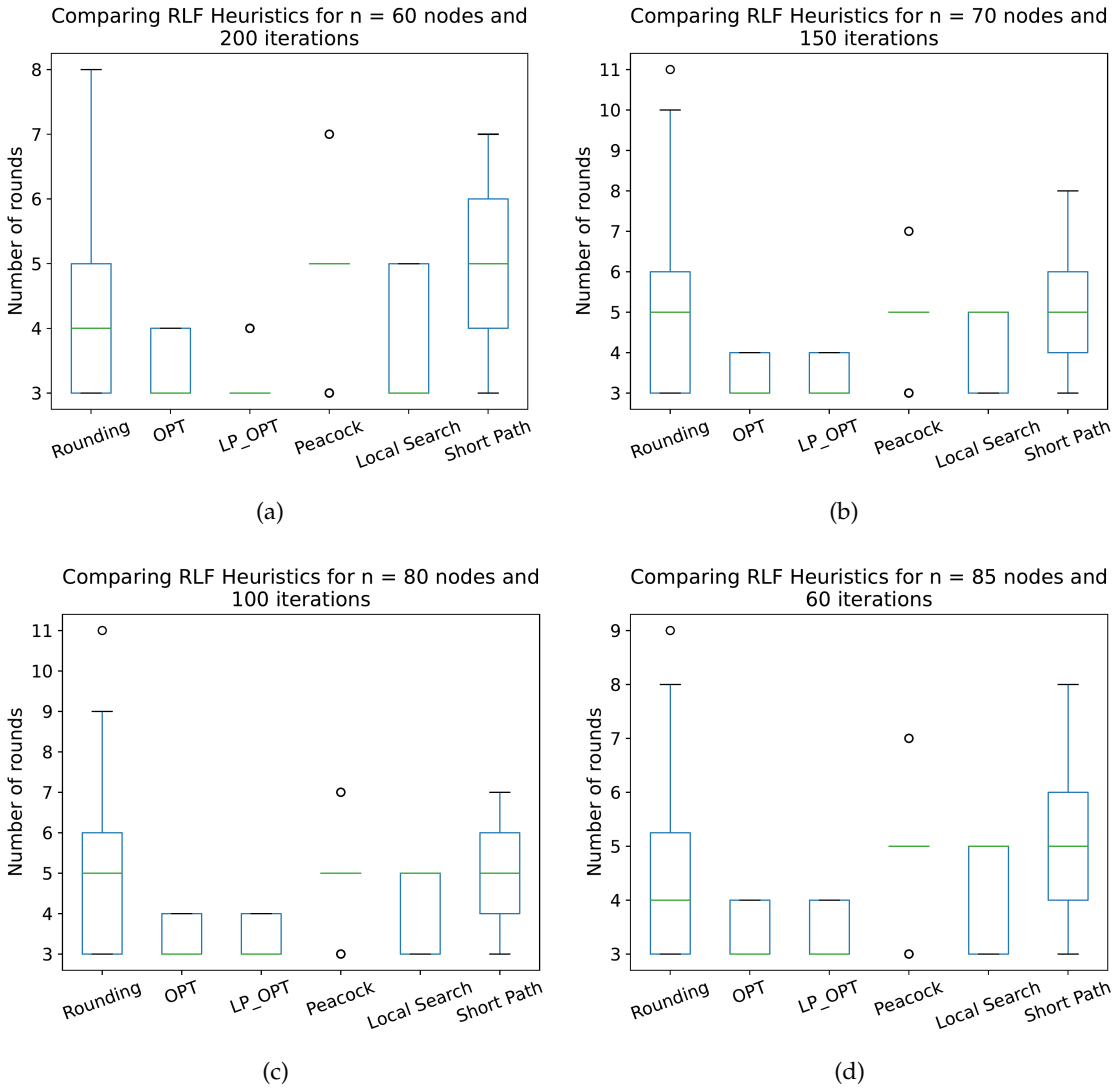


Figure 4.7: RLF: Comparison of Algorithms with OPT

5 Conclusion and Further Work

We presented both exact and approximation algorithms for the SLF and RLF Problems. We also implemented all algorithms and published our code (`https://github.com/RaduVintan/sdn/`) to ensure that our results are reproducible and falsifiable. For the SLF Problem, simple approaches like Greedy and Local Search seem not to work. Finding a suitable LP relaxation is also non-trivial. We reached an elegant Parameterized LP in Section 3.3.2 which allows (under some optimizations) to solve the problem both exactly (under an ILP formulation) and approximately (under an LP formulation and a suitable rounding algorithm, see Section 3.3.3) for a significant number of nodes. Our algorithm provides a way to compute the optimum much faster than with any other (known to us) method and the approximation quality of the LP Rounding seems to be very good in practice.

For the RLF Problem, which relaxes the SLF Problem, the LP approach again leads to encouraging results. The ILP formulation can be used to compute the optimum for large instances, and it is much faster than the previous ILP formulation [7] (adapted from [14]). However, if only an approximation is required, the best-performing heuristic is the Local Search algorithm. This heuristic improves on the well-known Peacock algorithm [5, 7]: it explicitly searches for the best set of forward edges to update. We also closed an open question from [7] by showing in Theorem 2 that Peacock is not an $o(\log n)$-approximation. For the proof, we exploited the fact that there exist graphs for which Peacock requires logarithmically many rounds. The construction which proves this fact in [7] turned out to be not fully correct. We identified the mistake and repaired the construction in Section 4.1.3. In Section 4.1.1 we built a preliminary framework which allows understanding all our non-LP based heuristics (including Peacock) from an elegant unified perspective (Algorithm 5).

On the negative side, we could not prove non-trivial approximation guarantees for our algorithms. In particular, it remains unknown whether there exists a non-trivial approximation for SLF and an $o(\log n)$ approximation for RLF. This is certainly the most important further work worth pursuing. Our best efforts of trying to find such proofs left us with the impression that

most current techniques might be not powerful enough to allow a direct attack. We suspect that the method of Iterative Rounding [19] might be useful for analyzing the performance of the Rounding LP algorithms. On the other hand, trying to prove hardness of approximation results for any of the two problems also seems to be a very reasonable endeavor.

List of Figures

Bibliography

[1] K.-T. Foerster, S. Schmid, and S. Vissicchio. *Survey of Consistent Software-Defined Network Updates*. In: *IEEE Communications Surveys & Tutorials* 21.2 (2019), pp. 1435–1461.

[2] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. *Abstractions for Network Update*. In: SIGCOMM '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 323–334.

[3] S. Fayazbakhsh, V. Sekar, M. Yu, and J. Mogul. *FlowTags: enforcing network-wide policies in the presence of dynamic middlebox actions*. In: Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14), Aug. 2013, pp. 19–24.

[4] R. Mahajan and R. Wattenhofer. *On Consistent Updates in Software Defined Networks*. In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. HotNets-XII. College Park, Maryland: Association for Computing Machinery, 2013.

[5] S. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid. *Transiently Consistent SDN Updates: Being Greedy is Hard*. In: *International Colloquium on Structural Information and Communication Complexity*. July 2016, pp. 391–406.

[6] A. Ludwig, J. Marcinkowski, and S. Schmid. *Scheduling Loop-Free Network Updates: It's Good to Relax!* In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. 2015, pp. 13–22.

[7] K.-T. Foerster, A. Ludwig, J. Marcinkowski, and S. Schmid. *Loop-Free Route Updates for Software-Defined Networks*. In: *IEEE/ACM Transactions on Networking* 26.1 (2018), pp. 328–341.

[8] D. Williamson and D. Shmoys. *The Design of Approximation Algorithms*.

[9] D. Bertsimas and J. Tsitsiklis. *Introduction to Linear Optimization*.

[10] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. URL: `https://doi.org/10.5281/zenodo.3509134`.

[11] W. McKinney. *Data Structures for Statistical Computing in Python*. In: *Proceedings of the 9th Python in Science Conference*. Ed. by S. van der Walt and J. Millman. 2010, pp. 56–61.

[12] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2022. URL: `https://www.gurobi.com`.

[13] A. A. Hagberg, D. A. Schult, and P. J. Swart. *Exploring Network Structure, Dynamics, and Function using NetworkX*. In: *Proceedings of the 7th Python in Science Conference*. Ed. by G. Varoquaux, T. Vaught, and J. Millman. Pasadena, CA USA, 2008, pp. 11–15.

[14] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid. *Transiently Secure Network Updates*. In: *SIGMETRICS Perform. Eval. Rev.* 44.1 (June 2016), pp. 273–284.

[15] Johansen, Nicklas S. and Kær, Lasse B. and Madsen, Andreas L. and Nielsen, Kristian Ø. and Srba, Jiří and Tollund, Rasmus G. *Kaki: Concurrent Update Synthesis for Regular Policies via Petri Games*. In: *Integrated Formal Methods*. Springer International Publishing, 2022, pp. 249–267.

[16] S. Schmid, B. C. Schrenk, and A. Torraiba. *NetStack: A Game Approach to Synthesizing Consistent Network Updates*. In: IFIP Networking Conference, 2022.

[17] N. Christensen, M. Glavind, S. Schmid, and J. Srba. *Latte: Improving the Latency of Transiently Consistent Network Update Schedules*. In: *SIGMETRICS Perform. Eval. Rev.* 48.3 (Mar. 2021), pp. 14–26.

[18] J. McClurg, H. Hojjat, P. Černý, and N. Foster. *Efficient Synthesis of Network Updates*. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2015.

[19] L. C. Lau, R. Ravi, and M. Singh. *Iterative Methods in Combinatorial Optimization*.