Computing Minimal Update Sequences for Graceful Router-wide Reconfigurations

Francois Clad[1], Stefano Vissicchio[2], Pascal Mérindol[1], Pierre Francois[3] and Jean-Jacques Pansiot[1]

[1]Université de Strasbourg
{fclad,merindol,pansiot}@unistra.fr

[2]Université catholique de Louvain
stefano.vissicchio@uclouvain.be

[3]Institute IMDEA Networks
pierre.francois@imdea.org

***Abstract*—Manageability and high availability are critical properties for IP networks. Unfortunately, with link-state routing protocols commonly used in such networks, topological changes lead to transient forwarding loops inducing service disruption. This reduces the frequency at which operators can adapt their network. Prior works proved that it is possible to avoid disruptions due to the planned reconfiguration of a link by progressively changing its weight, leading to a solution that does not require changing protocol specification.**

In this paper, we study the more general problem of gracefully modifying the logical state of multiple interfaces of a router, while minimizing the number of weight updates. Compared to single-link modifications, the router update problem is k-dimensional for a router having k neighbors. We also show that multi-dimensional updates may trigger new kind of disruptions that make the problem more challenging than the single link case. We then present and evaluate efficient algorithms that compute minimal sequences of weights enabling disruption-free router reconfigurations. Based on analysis of real IP network topologies, we show that both the size of such sequences and the computing time taken by our algorithms are limited.

I. Introduction

IP networks need to frequently undergo topological modifications, e.g., to support hardware replacement, software upgrades, and configuration updates [1], [2]. Those modifications can induce service disruptions, which, in turn, can reduce the ability of operators to frequently and reactively perform management operations [3] without affecting compliance to Service Level Agreements. Some ISPs have defined procedures to re-route the traffic out of a link [4] or a router [5] before shutting it down for maintenance. However, forwarding loops can still arise in spite of these procedures.

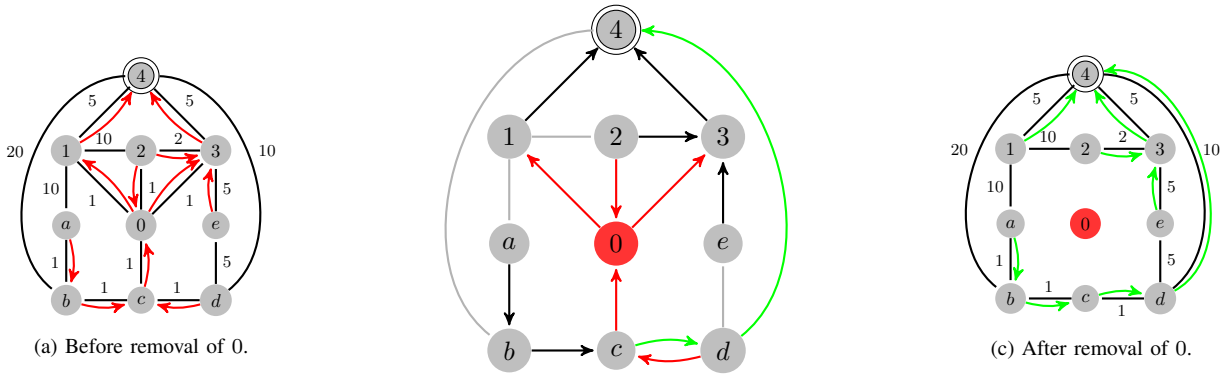
Typically, reconfigurations at a router granularity are among the most frequent operations typically performed in IP networks [6]. For this reason, we focus on the problem of supporting graceful reconfigurations in commonly used Link-State (LS) Interior Gateway Protocols (IGPs) such as OSPF [7] or IS-IS [8]. For the sake of simplicity, we focus on gracefully removing a node from an IGP network, e.g., in order to perform a software update on it without impacting traversing traffic. By symmetry, our techniques straightforwardly apply to the router addition case. Easy variants of our techniques can also be used to handle the reconfiguration of a subset of router links, e.g., for maintenance of single or multiple line-cards.

In OSPF and IS-IS, the state of each link is reliably flooded over the network. This way, each router can build a weighted graph representing the network in order to compute the forwarding paths as the shortest ones considering a given metric. Each topological modification involves a convergence process (i.e., to flood new LS information, recompute updated paths, and install corresponding FIB updates) during which transient forwarding loops may occur [9]. Such loops are due to inconsistent routing states among routers that possibly cause packet losses and delays increase [10], [11].

Fig. 1 exemplifies a transient loop in the case of a router removal. Namely, Fig. 1a and Fig. 1c represent the initial and final IGP topology, respectively, while Fig. 1b illustrates how inconsistent information held by different routers may cause transient forwarding loops during protocol convergence. The red and green arrows respectively represent the next-hops for destination 4 before and after the removal of node 0. Black arrows represent next-hops that remain the same. If router c updates its next-hop to 4 before d, then c starts forwarding traffic for destination 4 to d, while d keeps forwarding traffic to c. This creates a transient loop between c and d, which will only be solved when d also has updated its next-hop.

In this paper, we propose practical solutions that do not require changing any protocol specification, and apply to both symmetric and asymmetric link weights. Our solutions are based on a progressive modification of IGP link weights that induces a loop-free ordering in the update of all the forwarding paths. Practically, we split IGP convergence into multiple steps. During each step, weight modifications are flooded to all routers in the network through standard IGP Link-State Advertisements (LSAs). To reduce the operational impact, our algorithms minimize the number of convergence steps, hence limiting the additional control-plane overhead. Note that this marks a significant improvement of the naive strategy of applying graceful operations on a per link basis. Indeed, such a strategy can and typically do lead to long weight modification sequences [12]. To this end, we rely on the possibility of including weight changes for multiple links attached to a single router in a single LSA. Note that our solutions only modify the weight of links from the updated router to its neighbors such that the reconfiguration is entirely controlled by the updated

S. Vissicchio is postdoctoral researcher of the Belgian Fund for Scientific Research (F.R.S.-FNRS). This work has been partially supported by the European Community's Seventh Framework Programme (FP7/2007-2013) Grant No. 317647 (Leone).



(a) Before removal of 0.

(b) IGP convergence.

(c) After removal of 0.

Fig. 1: A transient forwarding loop can occur between nodes c and d for destination 4 during the routing convergence.

	Intermediate	
	Transient Loops	Forwarding Changes
with μloop delay	GBA	AGBA
w/o μloop delay	DGBH	

TABLE I: Overview of our algorithmic contributions.

router. In the example in Fig. 1, if node 0, before its removal, sends an LSA to update the weights of links $(0,1),(0,2)$ and $(0,3)$ to values 4, 2 and 4 respectively, then d will switch to its final forwarding path while c will not. This guarantee no transient loop between c and d when c updates its FIB.

Computing minimal weight modification sequences is challenging for three main reasons. First, all the destinations in the network must be taken into account, as our goal is to minimize the number of steps across them. Contrary to the link modification problem studied in previous work [13], the solution space for the node shutdown problem is k-dimensional, with k being the degree of the router to be updated. Second, applying several weight increments in a single LSA may lead to the use of next-hops that do not correspond to either initial nor the final ones. It is then not sufficient to analyze the initial and final routing states only to capture intermediate next hop changes that may provoke a new kind of disruptions such as flow deviations. In the example in Fig. 1, the updated node 0 may transiently use node 2 as next hop towards 4 during the convergence if the weights of 0 are set to the previously suggested values (4, 2 and 4, which are the minimal ones to avoid the loop between c and d). Third, those *intermediate forwarding changes* possibly lead to additional transient loops. In the initial state given in Fig. 1, one of the shortest paths from 2 to 4 includes 0, with 2 being in an Equal-Cost Multi-Path (ECMP) state. Hence, a new kind of transient loop can occur between 0 and 2, an intermediate loop that depends on the LSA injected to avoid the potential loop between c and d. Intermediate forwarding changes are necessary but not sufficient conditions to trigger such loops. Eventually, note that here a uniform increase of 3 on all links does not provoke such a loop while also avoiding loops between c and d. However, targeting minimal sequences generally comes at the cost of applying non-uniform weight-increment LSA.

To deal with transient loops (intermediate and non), we develop multiple algorithmic contributions, which are summarized in Table I. Our algorithms target two different settings.

In the first setting, the next-hops of the reconfigured router are kept constant during the entire IGP convergence by temporarily disabling synchronization of the router data-plane and control-plane, e.g., through the *μloop delay* feature [14]. In this setting, no intermediate transient loop can occur, and the graceful sequence minimization problem is optimally solved by the **Greedy Backward Algorithm (GBA)**. For every destination, GBA extracts the constraints that prevent transient loops resulting from the union of the initial and final forwarding paths. It then greedily computes minimal sequences of weight changes that verify all the extracted constraints.

In the second setting, data-plane and control-plane remain synchronized, and intermediate transient loops should be prevented algorithmically. For this setting, we propose two extensions of GBA. The first extension, called **Adjusted Greedy Backward Algorithm (AGBA)**, provably finds a minimal sequence that prevents both transient loops and intermediate forwarding changes. In particular, to avoid intermediate forwarding changes, it verifies that weight changes comply with a system of linear inequalities. Note that AGBA can also be used in the first setting if intermediate forwarding changes have to be avoided. The second extension, called **Dynamic Greedy Backward Heuristic (DGBH)**, computes sequences that prevent any kind of transient loops (including intermediate ones) but not intermediate forwarding changes in general. Intermediate transient loops are simply prevented by augmenting the set of constraints. DGBH is a heuristic in the sense that the sequences it computes are safe but not provably minimal with respect to the loop prevention problem.

In this paper, we consider the case of a non-urgent router update as for maintenance. Our approach can theoretically be combined with fast-reroute techniques to address failure use cases. However, investigating their practical interactions is out of the scope of this paper. We also assume that no network failure occurs while the sequence is applied on the network.

The rest of the paper is structured as follows. In Section II, we introduce the notation and we formalize transient loop constraints. In Section III, we describe our proposed solution, i.e., GBA, in the presence of μloop delay. In Section IV, we extend theory and GBA to algorithmically solve intermediate disruptions. In particular, we provide details on our AGBA and DGBH algorithms. In Section V, we report the results of our experiments performed on several real IGP network topologies. Our evaluation shows the effectiveness and efficiency of

our algorithms, suggesting the possibility of including them in current router OSes. Finally, in Section VI and VII, we compare our contributions with related works and conclude.

II. Model and Notations

In link-state IGPs, forwarding paths are computed as the shortest paths on a weighted graph $G = (N, E, w)$, such that N is the set of IGP routers, E is the set of IGP adjacencies between routers, and $w : E \to \mathbb{N}$ maps each directed link to its integer weight as defined by the IGP configuration. In the following, we focus on the problem of avoiding transient loops during the IGP convergence after a router removal. We denote the initial IGP graph as G, the router to be removed as 0, and the final IGP graph as $G' = G \setminus \{0\}$.

Since multiple paths can have the same cost for any source-destination pair, the set of shortest paths from each source to a single destination forms a Directed Acyclic Graph, called *Reverse Shortest Path DAG* (RSPDAG). Hence, we denote as $RSPDAG(d, X)$ the set of forwarding paths computed by IGP routers towards a given destination d in a graph X. Transient loops can occur during the transition from G to G' if and only if $RSPDAG(d, G) \cup RSPDAG(d, G')$ contains cycles (see, for example, Fig. 1). For the sake of simplicity, we use $RSPDAG(d)$ and $RSPDAG'(d)$ as shortcuts for $RSPDAG(d, G)$ and $RSPDAG(d, G \setminus \{0\})$ respectively.

Our convergence technique relies on a progressive modification of the IGP weights configured on the outgoing links of 0. Formally, it consists in computing a sequence of intermediate weighted graphs $G_0, G_1, \ldots, G_n$, where $G_0 = G$, $G_n = G'$, $\forall \in \{1, \ldots, n\}$, $G_i = (N, E, w_i)$, such that $\forall\, i \in \{0, \ldots, n-1\}$, $RSPDAG(d, G_i) \cup RSPDAG(d, G_{i+1}))$ contains no cycle. We generally refer to differences between weights in an intermediate graph $G_i \neq G$ and initial weights in G as *weight increments*, and we call a sequence $\{w_1, \ldots, w_n\}$ satisfying the previous property as a *weight increment sequence*. The term weight increment reflects our assumption that the weight of any link outgoing from 0 is always greater or equal to its initial weight. That is, since we aim at offloading traffic from the node to be removed, we do not consider sequences of weight modifications in which weights are decreased with respect to the initial state, as this can only make 0 more attractive. Nevertheless, we admit negative components in weight increments, e.g., if following positive increments.

We model a weight increment as a vector v, having $|v|$ components. For any weight increment v, a component $v[i]$ corresponds to the weight increment applied to the i-th outgoing link. Vectors of the same size can be compared, and partial order relationships can be defined between them. In particular, we say that two vectors v_1 and v_2 of size $k \geq 0$ are equal, i.e., $v_1 = v_2$, if $\forall i \in \{1, \ldots, k\}$ $v_1[i] = v_2[i]$. Similarly, $>$, $\geq$, $<$, $\leq$ relationships, hold on vectors if they hold on all the corresponding components. In addition, given two vectors v_1 and v_2 (such that $|v_1| = |v_2| = k$), we say that v_1 is *positively greater* than v_2, denoted $v_1 >^+ v_2$ if $\forall\, i \in \{1, \ldots, k\}$

$$\begin{cases} v_1[i] > v_2[i] & (\text{if } \; v_2[i] \in \mathbb{N}) \\ v_1[i] \geq 0 & (\text{if } v_2[i] \in \mathbb{Z}_{<0}) \end{cases}$$

We now define the concept of loop-constraint to formalize property of the weight increment sequence that must hold to avoid transient loops. More precisely, we define *loop-constraint*, or simply *constraint*, as the weight increment interval associated to a single loop. For any given transient loop L, a loop-constraint l is a vector pair $l := (\underline{l}, \bar{l})$. Vectors $\underline{l}$ and $\bar{l}$ have one component per outgoing link of router 0 (i.e., $|\underline{l}| = |\bar{l}| = k$ with k is the degree of router 0), and respectively represent the set of minimum and maximal weight increments that *prevents* L. To compute numerical values of loop-constraints, we rely on *delta vectors* Δ. Given a router $x \neq 0$ and a destination d, $\Delta_d(x)$ is the vector of weight increments such that the shortest paths from x to d include both the initial and final paths (as computed in G and G', resp.). Let $C'(x, d)$ be the cost of the shortest paths from x to d in G', l_i be the i-th link outgoing from 0, and $C(x, l_i, d)$ be the cost of the shortest path (without cycles) from x to d via l_i in G. By definition,

$$\Delta_d(x)[i] = C'(x, d) - C(x, l_i, d)$$

Let $\vec{0}$ be the all-zero vector. Then, the loop-constraint l associated to a loop L to a destination d is defined as

$$l := (\underline{l} := \min_{\forall x \in L}(\Delta_d(x)), \bar{l} := \max_{\forall x \in L}(\Delta_d(x)))$$

Note that, for any destination d, the set of vectors $\Delta_d(x) \forall x \in N$ is totally ordered. Indeed, for any router x, we have $C(x, l_i, d) = C(0, l_i, d) - C(0, d) + C(x, d)$. This implies that each x has the same offset among its $\Delta_d(x)$ components for each destination d. Moreover, note that $\Delta_d(x) = \vec{0}$ may imply an ECMP case on x potentially leading to an actual constraint.

By definition of Δ, the vector v_x verifying $\forall\; i \in \{1, \ldots, k\}$, $v_x[i] = \max(\Delta_d(x)[i] + 1, 0)$ is the smallest set of increments to be configured on the outgoing interfaces of router 0, such that router x switches to its final state and no longer uses 0 to reach d. Hence, in order to satisfy a loop-constraint l such that $\underline{l} = \Delta_d(z)$ and $\bar{l} = \Delta_d(y)$, an intermediate vector v must be positively greater than $\Delta_d(z)$, but not greater than or equal to $\Delta_d(y)$. Besides, if $v_x[i] = 0$ the weight of l_i can be arbitrarily increased. The distance of shortest paths from the node y to d verifying $\bar{l} = \Delta_d(y)$ is strictly shorter than the ones using l_i.

As an example of Δ and constraint vectors, consider again Fig. 1. In this figure, $\Delta_4(c) = (4\ 2\ 4\ -2)$ and $\Delta_4(d) = (2\ 0\ 2\ -4)$, where components respectively map to links $(0, 1)$, $(0, 2)$, $(0, 3)$, and $(0, c)$. As an illustration, $\Delta_4(c)[1] = 4$ since $C'(c, 4) = 11$ and $C(c, (0, 1), 4) = 7$. Adding 4 to the weight of link $(0, 1)$ would make the path from c to 4 through $(0, 1)$ as long as its final ones. Similar computations are applied to the other components of $\Delta_4(c)$ and $\Delta_4(d)$. According to those computations, forwarding paths from c (resp., d) are then ensured not to include 0 when weight increments greater than $\Delta_4(c)$ (resp., $\Delta_4(d)$) are applied to the outgoing links from 0. Moreover, the constraint l associated to the loop L between c and d is formalized as $l = (\underline{l} = \Delta_4(d), \bar{l} = \Delta_4(c))$.

By definition of l, applying weight increments positively greater than $\underline{l}$ (resp. $\bar{l}$) will cause the shortest paths from at least one router (resp. all the routers) in L not to traverse 0

anymore. In the previous example, applying a weight increment positively greater than $\underline{l} = \Delta_4(d)$ will cause d, but not necessarily c, to switch to its final shortest paths. Both c and d are guaranteed to switch to their respective final paths when the weight increments is positively greater than $\bar{l} = \Delta_4(c)$. To provably avoid a transient loop, we must then force weight increments changing only to forwarding paths of d, e.g. a relative increase of $(3\ 1\ 3\ 0)$, before applying the final weights.

To formally state the problem of finding such intermediate weight increments, we introduce the following terminology. We say that a weight increment v ***meets*** a constraint $(\underline{l}, \bar{l})$ if $v >^+ \underline{l}$ and $\exists\ i \in \{1, \dots, k\} \mid v[i] < \bar{l}[i]$. We also say that a weight increment v ***precedes*** a constraint l if $\exists\ i \in \{1, \dots, k\} \mid v[i] \leq \underline{l}[i] \neq 0$, and that v ***follows*** l if $\forall\ i \in \{1, \dots, k\} \mid v[i] \geq \bar{l}[i]$. Given a constraint l and a sequence of weight increments $\{v_0, \dots, v_n\}$ with $v_0 = \vec{0}$ (initial state of node 0) and v_n containing all ∞ (as after the removal of node 0), a pair of consecutive vectors v_i and v_{i+1} constitutes an *unsafe transition* if either i) v_i precedes $\underline{l}$ and v_{i+1} follows $\bar{l}$; or ii) v_i follows $\bar{l}$ and v_{i+1} precedes $\underline{l}$. Trivially, a pair of consecutive vectors is said to form a ***safe transition*** with respect to a given constraint if it is not unsafe. In the previous example, the sequence of relative increments $\{\vec{0}, \vec{\infty}\}$ contains an unsafe transition for the constraint $l = \{(2\ 0\ 2\ 0), (4\ 2\ 4\ 0)\}$. On the contrary, both transitions in $\{\vec{0}, (3\ 1\ 3\ 0), \vec{\infty}\}$ are safe with respect to l since the second vector $(3\ 1\ 3\ 0)$ meets l. Note that *MAX_METRIC* can be used in practice to enforce the final state $\vec{\infty}$.

A *safe sequence* contains safe transitions with respect to all loop-constraints. The following theorem holds.

Theorem 1. *A weight sequence s avoids a loop L if and only if s contains only safe transitions with respect to the constraint corresponding to L.*

Intuitively, Theorem 1 implies that, for each constraint $(\underline{l}, \bar{l})$, at least one vector must meet the constraint for each transition from weight increments smaller than $\underline{l}$ to those greater than $\bar{l}$, and vice versa. Intuitively, *always increasing* sequences seem to be the natural candidate for targeting minimality. A sequence $s = \{v_0, \dots, v_m\}$ is said *always increasing* if $\forall\ i \in \{1, \dots, m\},\ v_{i-1} \leq v_i$. Each sequence step meets a given subset of constraints cumulatively. A simplified version of Theorem 1 holds for always increasing sequences.

Theorem 2. *An always increasing weight sequence s avoids a loop L if and only if s contains at least one vector meeting the constraint corresponding to L.*

In this paper, we study the problem of finding minimal safe sequences with respect to all constraints. In particular, we present algorithms to compute always increasing sequences, that are provably minimal and safe. This also implies that restricting to always increasing sequences does not limit our ability to optimally solve the safe router update problem. In other words, for any network and for any router removal, at least one minimal safe sequence is always increasing.

III. Minimal Sequence Computation with GBA and μloop Delay

This section presents the Greedy Backward Algorithm, GBA, that we devised to support graceful router update in case μloop delay [14] is applied to the updated router 0. In this case, all the transient loops to be prevented can be extracted from the union of the initial and final RSPDAGs, since 0 postpones its next hops changes. They are called ***static constraints***. In practice, at each transition, 0 freezes its own convergence during a delay sufficiently large to ensure the convergence of its neighbors. It allows to avoid intermediate transient loops, that are equivalently local to 0, but not forwarding changes in general (see Sec. IV). Hence, the problem of computing minimal safe sequences can be formulated as follows.

Problem 1. *Minimal Loop-free Problem (MLP): Given a set $cs = \{(\underline{l}_1, \bar{l}_1), \dots, (\underline{l}_n, \bar{l}_n)\}$ of static loop-constraints, compute a minimal weight increment sequence which contains no unsafe transition for any constraint in cs.*

GBA is reported in Alg. 1. From a high-level perspective, the algorithm iteratively performs the following macro-steps:

I- Extract the *largest* constraint corresponding to a potential transient loop for each destination (**popMax**);

II- Backwardly compute a greedy weight increment that meets the maximum of extracted lower bound of constraints and update the set of constraints still to be met (**upMax**).

GBA stops when all the constraints are met.

We now provide more details on how the algorithm works. First, GBA is destination oriented, in the sense that it extracts constraints for each destination independently, from the merged DAG ($mdag$) obtained as the union of the initial (*RSPDAG*) and final (*RSPDAG'*) RSPDAGs. Before each intermediate vector computation, GBA only extracts the *last constraint* for each destination, i.e. the largest lower bound among the constraints associated to a destination. Second, GBA computes weight increments in a backward fashion, i.e. in the opposite order with respect to how they are to be applied. Using such a reverse order makes it possible to greedily build an update sequence of minimal length, as proved in [12]. Note that a greedy forward-based exploration of weight increments does not ensure minimality of the resulting sequence. This significant difference with previous works on graceful link operations [13] is due to an asymmetry in the way constraints may be satisfied: a vector v meets a constraint $(\underline{l}, \bar{l})$ if and only if $v >^+ \underline{l}$ and $v \not\geq \bar{l}$. While the first condition is a direct transposition of the scalar $>$, requiring each value in v to be greater than the value on same index in $\underline{l}$, the second condition allows all values but one to be greater than or equal to $\bar{l}$. The upper bound is far less restrictive than the lower one.

More precisely, GBA starts by computing the set of *affected destinations* as the nodes that are reached through 0 by at least one source (other than 0 itself). Indeed, if node 0 is not used by any source to reach a given destination, no transient loop could appear for that destination. Then, for each affected destination d, our algorithm computes *RSPDAG*(d), the initial forwarding graph towards d, while marking as *SRC* the subset of source nodes reaching d

Algorithm 1 Greedy Backward Algorithm

```
1: function GBA_MAIN(G, n)
2:     S = ∅
3:     for d in affectedDestinations (G, n) do
4:         dag, C, SRC = rspdag (G, d)
5:         dag', C' = rspdag (G \ n, d, SRC)
6:         mdag, list = INIT (dag, dag', SRC, n)
7:         mdag.popMax (list)
8:         if mdag.lc > 0 then
9:             for s in G.succ(n) do
10:                C(n, s, d) = w(n, s) + C'(s, d)
11:                mdag.offset[s] = C(n, s, d) − C(n, d)
12:            MDags.append (mdag)
13:    while MDags ≠ ∅ do
14:        v = 0⃗
15:        for mdag in MDags do
16:            for s in G.succ(n) do
17:                v[s] = max (v[s], mdag.lc − g.offset[s])
18:        S.append (v)
19:        for mdag in Mdags do
20:            m = ∞
21:            for s in G.succ(n) do
22:                m = min (m, v[s] + mdag.offset[s])
23:            if m < mdag.max then
24:                mdag.upMax (m)
25:                if mdag.lc = 0 then
26:                    MDags.remove (mdag)
27:    return S
```

Destination 1	$c_1 = \{a, b, a\}$	$S_1 =$	$\begin{pmatrix} 7 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 \end{pmatrix}$,
	$c_5 = \{b, c, c\}$		
Destination 2	$c_2 = \{c, d, c\}$	$S_2 =$	$\begin{pmatrix} 0 & 10 & 8 & 0 \end{pmatrix}$
Destination 3	$c_3 = \{c, d, c\}$	$S_3 =$	$\begin{pmatrix} 0 & 6 & 8 & 0 \end{pmatrix}$
Destination 4	$c_4 = \{c, d, c\}$	$S_4 =$	$\begin{pmatrix} 3 & 1 & 3 & 0 \end{pmatrix}$

GBA	
$c_1, c_4 \rightarrow c_2, c_3, c_5$	$S_{GBA} = \begin{pmatrix} 7 \\ 1 \\ 3 \\ 0 \end{pmatrix}, \begin{pmatrix} 9 \\ 10 \\ 8 \\ 0 \end{pmatrix}$

TABLE II: Destination and global sequences for the removal of node 0

through 0. This subset makes it possible to avoid useless calculations: GBA only focuses on the subgraph that may evolve due to the removal of node 0. Thus, the merged graph *mdag*(d), on which GBA detects cycles and their associated constraints, is computed as follows: *mdag*(d) = $G(SRC(d), E(RSPDAG(d) \bigcup RSPDAG'(d)) \bigcap (SRC(d) \times SRC(d))$. Δ values are then computed and associated to each node in *mdag*(d). At this stage, the **popMax** function checks whether transient loops could appear and, if so, computes the *last constraint*. If such a constraint exists, an offset value is then computed for each outgoing link of node 0. Otherwise, it means that no transient loop could possibly appear for this destination. This offset value reflects the *unattractiveness* of a link, and is equal to the difference of distance towards d through the associated link. Formally, we define *offset*$[d][x] = C(0, x, d) - C(0, d)$, where $C(0, x, d)$ represents the cost of the shortest elementary path in G from 0 to d through each successor x of 0. In the algorithm, we generalize for each node n in N (to provide sequences for all nodes $n = 0$). The purpose of such an offset is to avoid manipulating vectors when not necessary. Indeed, performing destination oriented operations does not require it since a total order exists among Δ for nodes in *SRC*. Eventually, the *mdag*(d) is added to the global *MDags* set.

Once the *MDags* set is computed, our algorithm enters the second phase. At each round of the global loop, a new greedy vector v is computed (and added to the sequence S) as the smallest one that is safe with respect to the *last constraint* for all subgraphs in the *MDags* set. Then, for each destination d, the actual distance update m associated to this vector is computed in order to make *mdag*(d) evolve accordingly. Note that a preliminary check is performed to know whether v could have an impact on *mdag*(d). If m is not lower than the maximum Δ value among the nodes in *mdag*(d), no constraint could have possibly been satisfied for d, so that it is not necessary to compute anything more for this destination. On the other hand, if m is lower than at least one Δ value in *mdag*(d), **upMax** is called (at least one node is impacted). This function modifies the graph, now considering v as the final weight assignment, and prunes all nodes that cannot be involved in any cycle. It then extracts the new *last constraint*, if any, and returns 0 otherwise. If there are no more constraints to be satisfied for this destination, it is removed from the *MDags* set. The main loop iterates this way until *MDags* is empty, meaning that all constraints are satisfied by the sequence S.

Table II gives the sequences obtained by running GBA on the graph described on Fig. 1, for each *affected destination* separately, and the global one. In this case, our algorithm provides a sequence that satisfies all loop constraints with only two intermediate updates. Formal properties and proofs demonstrating the safety and minimality of GBA are provided and generalized for AGBA in the appendix. We also provide algorithmic details for the internal procedures.

Let us now focus on the time performance of GBA. There exist several ways to efficiently implement GBA, which can be tuned for a particular deployment: inside a router or in a management tool. While minimizing the worst case complexity appears to be the main goal in the former case, the average complexity becomes prevalent when considering the latter.

In order to ensure an efficient computation, GBA implements several "pruning processes" that reduce the number and the size of the graphs to be considered. These improvements also limit graph manipulations to the strict necessary. First, the set of destinations to be considered by GBA is reduced only to the ones that are initially reached through the removed node. This is because increasing the weights configured on its outgoing links will not make this node more attractive for other destinations. An efficient way to compute this subset of destinations consists in computing an $SPDAG$ on each neighbor of 0 beforehand. The set of ***affected destinations*** corresponds to all nodes that are reached via 0 in such $SPDAG$. Second, the calculations performed by GBA for the remaining destinations are lightened thanks to a set of ***subgraph reductions***. First, source nodes that actually use node 0 to reach the destination are tagged when computing the initial routing graph. It allows to restrict further computations to this subset of nodes only. Second, for each destination, we maintain a variable containing the largest Δ value among the nodes that have not yet been concerned by previously backwardly computed greedy vectors: their Δ is lower or equal to the greedy vector. Thanks to this variable it is possible to check whether the next greedy vector has an impact on a given

destination $mdag$, and then limit each $mdag$ manipulations to useful cases. Finally, note that a *natural* Δ order exists among the nodes in an $mdag$ (notations are described in details in the appendix). Formally, for a given destination, we have $\Delta(\textbf{prePred(x)}) \leq \Delta(x)$ so that it is possible to limit the node exploration with a BFS algorithm on the subgraph G induced by initial forwarding states for nodes in SRC (defined with edges in $(x, \textbf{prePred(x)})$, the predecessors in *RSPDAG* (d)).

At the microscopic view, the core component of GBA we use is a cycle detection algorithm. It allows for initializing the constraint system and to extract the *last constraints* at each iteration of the main loop. This algorithm helps at two levels: first, it gives GBA the ability to definitively remove non relevant nodes and edges as soon as a given weight assignment removes them from the constraint system; second, it can be repeatedly applied on a clone of the remaining graph in order to extract new *last constraints*. This constraint extraction mechanism has a complexity of $|E|$ and is never called more than once for each node in an *mdag*.
Each procedure of GBA comes with a specific complexity:

- Last constraints extraction has a cost of $O(|N| \times |E|)$. Note that the *RSPDAG* computation has a complexity of $O(|N| \times (|N|log_2(|N|) + |E|))$;
- The number of iterations of the main loop can be limited to a given length parameter $p << |N|^2$ (p being the targeted maximal sequence size). Inside the loop we have:
  - Vector manipulations for all destinations with a complexity of $pk|N|$ (k being the degree of node 0);
  - The constraints update comes at a cost of $O(min(p \times |N| \times |E|), |N^2| \times |E|))$ for all destinations.

Eventually, GBA has a worst case complexity in $O(|N|^4)$ if node 0 has a degree of $k = |N|$ (or if $|E| \approx |N|^2$ in general). However, in practice it is worth noticing that p can be picked as an arbitrary low value such as $p \leq 5$ to limit the complexity of GBA to $O(|N|^3)$.

IV. Algorithmic Extensions for Preventing Disruptions due to Intermediate Updates

Applying non-uniform weight modifications on the outgoing links of node 0 allows for minimizing the length of an increment sequence. Indeed, using different weight increments over several outgoing links of 0 can help to satisfy a subset of constraints for different destinations in the same update step. In some cases, there is no equivalent uniform weight increase step. Unfortunately, such modifications can introduce new intermediate disruptions, namely intermediate forwarding changes and intermediate transient loops around 0.

In the following, we illustrate and describe those disruptions and how to deal with them within the GBA algorithm. In Sec. IV-A, we describe the Adjusted Greedy Backward Algorithm (AGBA) that computes provably minimal sequences preventing all kinds of intermediate disruptions. In Sec. IV-B we present an algorithmic alternative to the μloop delay feature, called DGBH, preventing all intermediate transient loops at the cost of slightly longer sequences.

Fig. 2 depicts the shortest paths on the network in Fig. 1 when applying the first vector, $(7, 1, 3, 0)$, computed by GBA

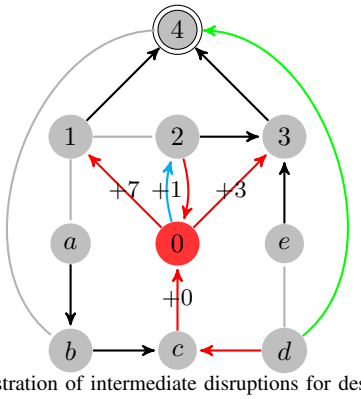


Fig. 2: Illustration of intermediate disruptions for destination 4.

for the removal of node 0. Aside from forcing node d to shift to its final path, this weight increment also makes 0 update its shortest paths to 4. More precisely, 0 starts using nodes 2 and 3 instead of 1 and 3 as next-hops, and forwarding traffic on path $(0\ 2\ 3\ 4)$, that it does not use either in G or in G'. Note that, contrary to final paths that are expected to be used after the modification, such an intermediate path may not be sufficiently provisioned, hence leading to congestion. In this example, node 3 may act as a bottleneck on the paths used by 0 to 4, which are no longer disjoint. Even worse, a transient loop can occur between 0 and 2, since 2 was initially using 0, as highlighted by the red arrow from 2 to 0.

In the following, we will refer to any set of forwarding paths used by 0 after the application of weight increments and not coinciding with both its initial and final set of paths as *intermediate forwarding change*. Beyond increasing the risk of congestion, intermediate forwarding changes can translate to experiencing multiple temporary paths between some source-destination pairs before stabilizing on the final ones. Depending on the latency of each intermediate path with respect to the initial and final ones, this may increase the probability of out-of-order packet delivery, delay and TTL variations during the IGP convergence.

All those variations may have a negative impact on control mechanisms implemented at the transport layer.

Intermediate forwarding changes can cause *intermediate transient loops*, as the loop between 0 and 2 in the example in Fig. 2. Those loops depend on the shortest paths on intermediate forwarding graphs obtained by applying non-uniform weight increments. As such, they do not correspond to cycles in the graph $RSPDAG \cup RSPDAG'$, with $RSPDAG$ and $RSPDAG'$ being the initial and final RSPDAGs to a destination d, respectively. Note that these loops always include node 0. These loops induce two complications. First, intermediate transient loops are not captured by GBA, as shown by the example in Fig. 2. Second, they map to *dynamic constraints* depending on the increment sequence itself (as opposed to the GBA constraints that can be computed through a static analysis on the initial and final RSPDAGs).

A. Avoiding Intermediate Forwarding Changes with AGBA

Since the root cause of intermediate nexthops leading to loops and new forwarding paths is induced by forwarding changes on node 0, a sufficient and necessary condition to

Algorithm 2 AGBA-1 : Minimal Constraints Initialization

```
1: Minimal constraints matrix M = ∅
2: Initial successors subset S*
3: for d in N do
4:     S* = ∅
5:     for x in n.succ() do
6:         offset[d][x] = w(n, x) + C'(x, d) − C(n, d)
7:         if offset[d][x] = 0 then
8:             S*.append(x)
9:     for s in S* do
10:        for x in n.succ() do
11:            M[s][x] = min (M[s][x], offset[d][x])
```

Algorithm 3 AGBA-2 : Greedy Vector Adjustment

```
1: function ADJUST_VECTOR(n, M, gv)
2:     indexes = n.succ()
3:     while indexes ≠ ∅ do
4:         p = pop_max_index (indexes, gv)
5:         for x in n.succ() do
6:             if M[p][x] = 0 then
7:                 gv[x] = gv[p]
8:             else if gv[x] ≤ gv[p] − M[p][x] then
9:                 gv[x] = gv[p] − M[p][x] + 1
10:    return gv
```

avoid any intermediate edge consists in enforcing that 0 *maintains its next-hops* throughout the IGP convergence.

Consistently with the rest of the paper, we denote the initial IGP graph as G. We also denote the component of a vector v associated to a link $(0, x)$ as $v[x]$.

Definition 1. *A node s is called **initial successor** of 0 to d if $(0, s)$ is the first edge of a path in $RSPDAG(d, G)$. We denote the set of initial successors of 0 to d as $S^*(d)$.*

Intuitively, initial successors are next-hops used by 0 to reach d in G, In the example in Fig. 2, nodes 1 and 3 are initial successors of 0 for destination 4, while 2 and c are not.

Definition 2. *Let d be a destination, s^* be an initial successor of 0 to d, and v be a weight increment. We define the* intermediate forwarding Change Prevention Conditions (CPCs) *as the set of inequalities*

$$v[s] = v[s^*]$$

$$v[x] > v[s^*] - offset[d][x]$$

for each initial successor $s \in S^*(d)$ of 0, and for each other neighbor x of 0 such that $x \notin S^*(d)$.

As an illustration, consider again Fig. 2 and let $s^* = 1$. The CPCs for destination 4 consists of inequalities $v[1] < v[2] + 2$ and $v[1] = v[3]$. Observe that CPCs are formulated with respect to a single initial successor (i.e., 1 in the example above). However, the correctness of the CPCs does not depend on the considered initial successor.

Moreover, for each neighbor $x \notin S^*(d)$, it must be $C(0, d) < C(0, x, d)$ by definition of initial successors. Hence, $offset[d][x] > 0$, and the following property holds.

Property IV.1. *Any CPC inequality can be written as $v[s^*] \leq v[x] + m$, with $m \geq 0$.*

Intuitively, CPCs impose that, for a given destination, paths via initial successors of 0 should be shorter than any other paths via a non initial successor (i.e., we aim to adjust their increments such that they do not results in intermediate shortest paths). Hence, verifying CPCs for a destination d guarantees that the shortest paths from 0 to d remain the same. This implies the following theorem (whose proof is reported in the Appendix).

Theorem 3. *If a weight increment v satisfies the CPCs for all destinations, no forwarding change occurs when v is applied.*

Since intermediate transient loops cannot occur in the absence of forwarding changes, the following corollary holds.

Corollary 1. *If a weight increment v satisfies the CPCs for all destinations, no intermediate transient loop occurs.*

We now show a GBA generalization, called Adjusted GBA or AGBA, that guarantees prevention of intermediate edges by enforcing accommodation of CPCs for all network destinations. More precisely, AGBA solves the following problem.

Problem 2. *Minimal intermediate Change-free and Loop-free Problem (MCLP): Given a set $\mathcal{C}$ of loop-constraints and a set $\mathcal{A}$ of CPCs, compute a minimal weight increment sequence that contains no unsafe transition for any constraint in $\mathcal{C}$, and no weight increment that violate any condition in $\mathcal{A}$.*

Provided that all the loop-constraints and the CPCs are correctly enumerated, solving an MCLP instance implies preventing all possible convergence loops and forwarding changes in the corresponding network as per Theorems 1 and 3.

To solve the MCLP problem, at each iteration, AGBA post-process each weight increment gv as computed by GBA. To this end, AGBA adds two main algorithmic steps to each iteration of GBA. One in its initialization, the other within the main loop iteration to adjust the greedy vector.

First, AGBA computes every offset values and optimizes them across all destinations, as shown in Alg. 2. In particular, for each destination, it computes all the offsets and identifies the initial successors (see lines 5–8 in Alg. 2). Moreover, for each pair initial successor and neighbor of 0, it only keeps the smallest offset (see lines 9-11 in Alg. 2), as it corresponds to the most constraining CPCs.

Second, AGBA modifies the greedy vector gv as computed by GBA, applying the following operations. 1) *vector sorting*, in which the components of gv are considered from the biggest to the smallest one (this corresponds to consider all the CPCs in decreasing order). The goal is to retrieve the up to date pivot component p (line 4 in Alg. 3); and 2) *vector adjusting*, in which the current component of gv is modified to satisfy all the sorted CPCs. AGBA enforces the CPCs by imposing:

$$v[s] = m_d$$

$$v[x] = m_d - offset[d][x] + 1$$

where $s \in S^*(d)$, $x \notin S^*(d)$, and $m_d = max_{\forall s \in S^*(d)}(v[s])$. That is, given a weight increment, AGBA calculates the maximum component corresponding to an initial successor, which we call *pivot component*, and imposes that all the other components of the vector must enforce the CPCs with respect to such a pivot component. Consider again the example in

Fig. 2. The pivot component of the shown weight increment v is $v[1]$ and $m_4 = 7$. AGBA imposes that $v[1] = v[3] = 7$, $v[2] = 6$ and $v[c] = 2$. Eventually, the complete sequence computed by AGBA on the network in the figure is

$$S_{AGBA} = \left\{ \begin{pmatrix} 3 \\ 2 \\ 3 \\ 3 \end{pmatrix}, \begin{pmatrix} 7 \\ 6 \\ 7 \\ 7 \end{pmatrix}, \begin{pmatrix} 8 \\ 7 \\ 8 \\ 8 \end{pmatrix}, \begin{pmatrix} 9 \\ 10 \\ 9 \\ 9 \end{pmatrix} \right\}$$

which is two increments longer than the GBA one but one lower than the uniform sequence $\{3, 7, 8, 9, 10\}$. While it may appear as a large sequence increase in practice, our experiments reported in Sec. V show that the sequence increase is not significant in many realistic cases.

To conclude, the following theorems show that AGBA finds minimal increment sequences solving the MCLP problem. Proofs are included in the Appendix.

Theorem 4. *For any MCLP instance $I =< \mathcal{C}, \mathcal{A} >$,* AGBA *always terminates in $O(|\mathcal{C}|)$ iterations.*

Theorem 5. *The weight sequences computed by* AGBA *prevent both transient loops and forwarding changes.*

Theorem 6. AGBA *computes a minimal sequence solving any given MCLP instance.*

Intuitively, AGBA is correct and optimal because CPC constraints are statics in the same vein as transient loops ones. The greedy behavior of GBA is then still ensured with respect to an additional kind of static constraints, i.e., the "minimal" resolution of a linear inequation system.

B. Avoiding Intermediate Loops with DGBH

AGBA enforces strong consistency guarantees during IGP convergence at the cost of increasing the sequence length. In the following, we explore a different trade-off between consistency guarantees and sequence length. In particular, we investigate an algorithm that prevents transient loops but not necessarily intermediate forwarding changes, i.e., solving the following problem.

Problem 3. *Minimal Intermediate Loop-free Problem (MILP): Given a set $\mathcal{C}$ of loop-constraints, compute a minimal weight increment safe sequence that does not result in any intermediate transient loop on* 0.

Since the MILP problem allows 0 to change its forwarding paths during the application of the increment sequence, we now face dynamic loop constraints, i.e., depending on the sequence being computed. In order to deal with those constraints, our greedy heuristic, called DGBH, potentially adds loop constraints at each iteration. In practice, it simply extends GBA to enable UPMAX, POPMAX and their related cycle detection functions to retrieve cycles including node 0 before the computation of each greedy vector. This way it computes lower bounds of last constraints related to intermediate loops. Note that those additional operations require neither any extra information nor dedicated computation process, keeping a time efficiency similar to the original GBA.

While sequences computed by DGBH are correct, they are not guaranteed to be minimal. Consider again the network in Fig. 1. DGBH computes the following sequence.

$$S_{DGBH} = \left\{ \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \\ 3 \\ 0 \end{pmatrix}, \begin{pmatrix} 7 \\ 3 \\ 5 \\ 0 \end{pmatrix}, \begin{pmatrix} 9 \\ 10 \\ 8 \\ 9 \end{pmatrix} \right\}$$

Vector $(1\ 0\ 1\ 0)$ avoids the intermediate loop between 0 and 2. The given sequence is not minimal with respect to MILP. Indeed, consider the case in which $(3\ 3\ 3\ 0)$ is used as second vector, preventing 0 to change its path (using an approach similar to the AGBA one). This vector would have prevented both the loop between c and d, and the intermediate one between 0 and 2, hence leading to a loop-free sequence with 3 metric increments. Nevertheless, adjusting vectors to prevent 0 to change its path forces some solutions of the MILP problem to be discarded, which can in turn lead to non-minimal sequences in other cases. Even worse, such adjustments can induce new intermediate loops while preventing some others.

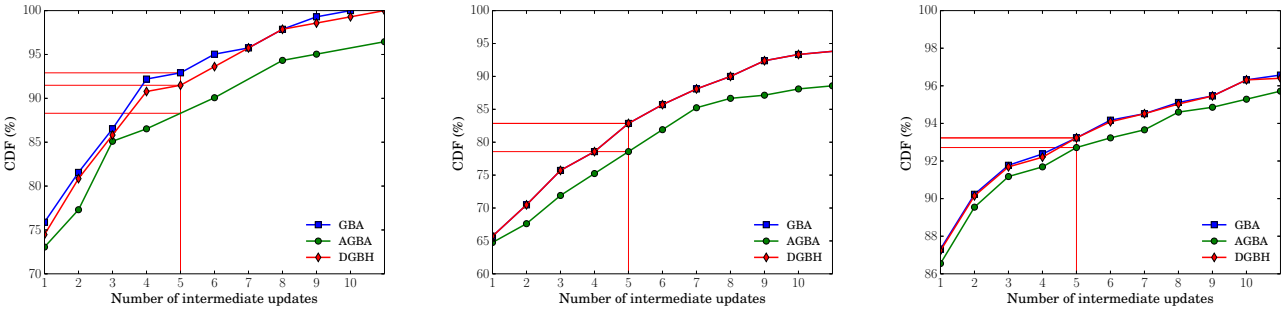
Generally speaking, from a GBA-based perspective, two strategies can be adopted to prevent intermediate loops, namely, 1) modify the current greedy vector to avoid the intermediate change at 0; OR 2) add a constraint to the computation of the next greedy vector, to force another node participating in the loop to not use 0 before it switches. Unfortunately, none of the two strategies always leads to minimal sequences when applied independently. While the presence of alternative strategies at each step seems to force a combinatorial space exploration, the theoretical problem of efficiently solving MILP is left open. However, our evaluation (see Sec. V) shows that a heuristic based only on the second strategy, i.e., DGBH, computes sequences as short as GBA in the vast majority of our experiments.

V. EVALUATION

In this section, we evaluate the performance of our algorithms based on real IP networks, using the actually configured link weights. Our evaluation criteria are twofold. We first present the length of weight sequences resulting from our three algorithms, for the case of node removal. Second, we evaluate the computing time gains of our implementation improvements to GBA, on a very large IGP network.

We evaluated our algorithms on a wide set of IGP network graphs of various shapes and sizes. However, due to space limitations, we chose to restrain the presented results to only our three largest topologies. These are real Tier-1 and Tier-2 networks that we anonymized for confidentiality reasons (the number of nodes and edges are also rounded). Sequence length evaluation for smaller networks, as well as comparison with the single-link solution can be found in [12]. We performed those experiments by running a freely available C implementation of our algorithms[1], on common hardware (2.5 GHz CPU and 4GB RAM) and software (Debian 7).

[1] http://sourceforge.net/projects/metric-incr/



(a) ISP1 (140 nodes, 410 edges) (b) ISP2 (210 nodes, 780 edges) (c) ISP3 (1200 nodes, 4000 edges)

Fig. 3: Sequence length distribution. DGBH has no significant impact on sequence lengths, while AGBA produces slightly longer sequences.

A. Sequence length

On Fig. 3, we show the proportion of node removal operations that could be safely performed with at most x intermediate updates, using our different algorithms. In particular, we highlight the proportion of sequences containing at most 5 intermediate updates with each algorithm. Note that the length of the sequence has a linear influence on the time an operator has to wait before actually performing the planned operation. In the case of a maintenance event, we assume that the operation can be slightly delayed as we choose a sequence length of 5 as an arbitrary, yet realistic, upper bound.

On ISP1 (Fig. 3a), 5 intermediate vectors are sufficient for almost 93% of the nodes to be safely removed if a data plane freezing mechanism is available. If not, as many intermediate vectors are still enough to prevent any transient loop for 91.5% of the nodes (DGBH) or all possible disruptions for more than 88% of them (AGBA). On ISP2 (Fig. 3b), 83% of the nodes can be removed with no transient loop using 5 intermediate vectors or less, whether or not a data plane freezing solution is available. Disruption freeness is ensured in 78.5% of the cases. This proportion of nodes requiring a *short enough* sequence is even larger on ISP3 (Fig. 3c), with more than 93% for GBA and DGBH, and 92.7% for AGBA. Overall, these figures show that, whatever the algorithm, most node removal operations could be safely performed with only a few intermediate updates. Intuitively, this is due to the high connectivity of typical network designs, as it tends to limit potential loops to the neighbors of the node to be removed. Consequently, both the number of loop constraints and sequence size lengths are much smaller than their theoretical upper bounds.

The second interest of this figure is to evaluate the overhead, in terms of sequence length, of the algorithms handling local disruptions, namely DGBH and AGBA, compared to standard GBA. We notice that DGBH, even if not providing guarantees of minimality, generally produces sequences of almost the same length as GBA. Besides, disruption free sequences obtained with AGBA are only slightly longer, hence permitting to do without a data plane freezing mechanism at a slight cost.

B. Computing time

ISP3 being by far our largest evaluation topology, we chose to present computing time measurements focusing on this graph. Note that computing times on smaller topologies, such as ISP 1 and 2, are of a few milliseconds.

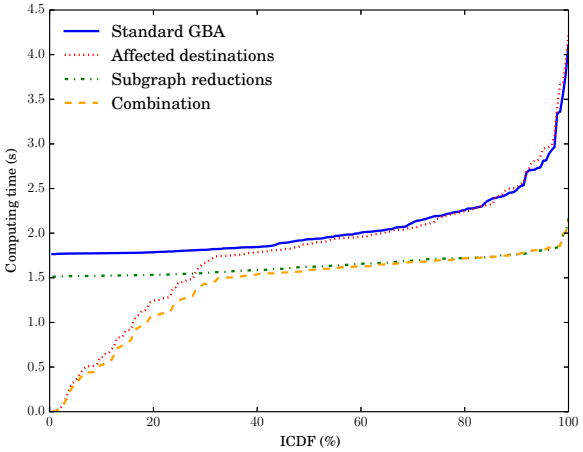


Fig. 4: Computing time distribution on ISP3

On Fig. 4, we evaluate the time required to compute a sequence for the removal operation of all nodes in the graph, and the practical efficiency of our algorithmic improvements. Although these results are obtained using standard GBA, opting for another variation (DGBH or AGBA) has no significant effect on the computing times (time curves are superposed).

With a basic GBA implementation (solid blue line), computing such sequences usually takes from 1.75 to 2.5 seconds, but may require up to 4 seconds for worst cases. However, these computing times can be dramatically reduced by implementing a few algorithmic improvements. On the first hand, we can reduce the set of destinations to be considered by GBA only to *affected destinations*. This results in a strong decrease of the computing time for more than a third of the nodes. However, for nodes having a large degree and being used to reach most of the destinations in the network, the cost of the preliminary phase may exceed the benefits it provides, having thus slight negative effects in some cases. On the second hand, the time required to compute a sequence in worst cases can be almost halved by re-using intermediate results and limiting redundant calculations within GBA subfunctions. These *subgraph reductions* additionally reduce the computing time for all nodes by about 250 ms. Eventually, both improvements can be used together in order to combine their positive effects (*combination*). Indeed, the drawback of computing affected destinations fades away when it is associated with the other subgraph improvements.

From our evaluation, we can conclude that our algorithms

generally produce sequences of limited length, regardless of the GBA variation, in a much reasonable time. These results encourage us to consider the use of this solution on production routers. In the case of planned operations, transient inconsistencies could thus be avoided at the cost of a slightly extended convergence time.

VI. Related Works

Protocol based solutions have been investigated in the past, notably oFIB [15]. oFIB is a proposal to order the FIB update in rerouting routers in a way that ensures forwarding consistency during the convergence process. oFIB can prevent loops in the case of link and node shutdown, as well as during corresponding up events. However, the networking community acknowledged that such a solution requires complex modifications to the specification of OSPF and ISIS, and require support in all routers of the network in order to be applicable. The technique presented in this paper achieves the same goal (handling both link and node reconfigurations) without protocol modification.

An alternate solution, PLSN, is described in [16]. It achieves loop avoidance one hop away from a rerouting router, by controlling the time at which the rerouting router updates its FIB. With PLSN, a rerouting router delays, by a fixed amount of time, a FIB update for a given destination if its new next-hop is not loop-free for the destination. PLSN can be seen as a reduced version of oFIB that does not require router-to-router synchronization, at the cost of reduced applicability.

Hitless network migration techniques such as [17] could be considered as alternatives to the solution described in this paper. They are however focused on network-wide changes, as they require to temporarily maintain two IGP configurations in the whole network, and switch from the initial to the final one on a per router basis. As such, it implies higher management overload, and longer reconfiguration processes. The technique explained in this paper reduces the operational impact of single router reconfigurations.

Similarly, operations on single routers can be gracefully performed by applying safe single-link techniques [13] sequentially on each link maintained by the router. However, our evaluation shows that such an approach is far from being optimal from a reconfiguration duration point of view.

Solutions have also been investigated for the case of routing software upgrades on recent router architectures, which are able to fully dissociate the routing and forwarding engines. The *I'll Be Back* capability [18] allow the router to continue forwarding packets even if the routing process is down, while preventing possible forwarding loops. Our approach enables to solve the same problem (graceful router software upgrades) without requiring modifications of current routing protocols and with minimal control-plane overhead. This however comes at the cost of local traffic shifts.

This paper extends a preliminary version of our work that appeared in [12]. Our extensions include (i) algorithmic details of GBA, which help to understand how the algorithm can be implemented efficiently; (ii) proposal of a generalization of GBA, AGBA, that optimally solves the problem of avoiding any disruption in the absence of local delay; (iii) generalized proofs, showing correctness and optimality of both GBA and AGBA for their respective problems; and (iv) wider evaluation of our algorithms, including time efficiency analysis.

Our algorithms for graceful router-wide updates remain safe in networks with multiple routing domains connected through route redistribution [19].

Recent works [20] show that the application of IGP weight increments can trigger anomalies in BGP, due to its interaction with IGP. Further, it presents sufficient conditions that guarantee BGP anomalies not to occur. In this paper, we implicitly assumed those conditions. In particular, we rely on ingress-to-egress packet encapsulation to avoid transient loops for BGP traffic. The numerous benefits of encapsulation in transit networks and the fact that it is commonly used by service providers make this assumption realistic.

VII. Conclusions

In this paper, we described techniques to support graceful router updates in link-state routing protocols. Our techniques do not require changes to IGP specifications and are based on efficient algorithms finding minimal sequences of link weight increments that avoids transient forwarding loops. While we focused on router removal, our techniques can also address other use cases, like router addition (by applying the computed sequence in the reverse order), and arbitrary sets of weight increase or decrease on links maintained by a given router (by applying part of the sequences computed by our algorithms).

We first presented the GBA algorithm, which is correct and optimal when used in conjunction with local delay [14]. We then introduced AGBA, a generalized version of GBA that computes minimal sequences avoiding the use of intermediate paths. AGBA does not require local delay. Furthermore, we discussed a heuristic, DGBH, aimed at providing shorter sequences than AGBA while still not requiring local delay.

By extensive evaluation, we showed the practicality effectiveness of our algorithms. Even on large Tier-1 networks, they need few seconds to compute safe weight increment sequences, which are likely shorter than 5 steps. Such time efficiency indicates the possibility of including our algorithms in current routers' software.

References

[1] P. Pongpaibool, R. Doverspike, M. Roughan, and J. Gottlieb, "Handling IP Traffic Surges via Optical Layer Reconfiguration," in *OFC*, 2002.

[2] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, Y. Ganjali, and C. Diot, "Characterization of Failures in an Operational IP Backbone Network," *IEEE/ACM Trans. Netw.*, vol. 16, pp. 749–762, August 2008.

[3] J. Martin and A. Nilsson, "On Service Level Agreements for IP Networks," in *INFOCOM*, 2002.

[4] R. Teixeira and J. Rexford, "Managing Routing Disruptions in Internet Service Provider Networks," *IEEE Communications Magazine*, vol. 44, no. 3, pp. 160 – 165, March 2006.

[5] D. McPherson, "Intermediate System to Intermediate System (IS-IS) Transient Blackhole Avoidance," IETF, RFC 3277, April 2002.

[6] A. Medem, R. Teixeira, N. Feamster, and M. Meulle, "Joint analysis of network incidents and intradomain routing changes," in *CNSM*, 2010.

[7] J. Moy, "OSPF Version 2," IETF, RFC 2328, April 1998.

[8] D. Oran, "OSI IS-IS Intra-domain Routing Protocol," IETF, RFC 1142, February 1990.

[9] H. Ito, K. Iwama, Y. Okabe, and T. Yoshihiro, "Avoiding Routing Loops on the Internet," *Theory of Computing Systems*, vol. 36, pp. 597–609, 2003.

[10] H. Pucha, Y. Zhang, Z. M. Mao, and Y. C. Hu, "Understanding Network Delay Changes Caused by Routing Events," *SIGMETRICS Performance Evaluation Review*, vol. 35, no. 1, pp. 73–84, June 2007.

[11] Y. Zhang, Z. Morley Mao, and J. Wang, "A Framework for Measuring and Predicting the Impact of Routing Changes," in *INFOCOM*, 2007.

[12] F. Clad, P. Merindol, S. Vissicchio, J.-J. Pansiot, and P. Francois, "Graceful Router Updates for Link-State Protocols," in *ICNP*, 2013.

[13] F. Clad, P. Merindol, J.-J. Pansiot, P. Francois, and O. Bonaventure, "Graceful Convergence in Link-State IP Networks: A Lightweight Algorithm Ensuring Minimal Operational Impact," *IEEE/ACM Transactions on Networking*, vol. 22, no. 1, pp. 300–312, February 2014.

[14] S. Litkowski, B. Decraene, and P. Francois, "Microloop prevention by introducting a local convergence delay," IETF, Internet-Draft, 2013.

[15] P. Francois and O. Bonaventure, "Avoiding transient loops during the convergence of link-state routing protocols," *IEEE/ACM Trans. Netw.*, vol. 15, no. 6, pp. 1280–1292, December 2007.

[16] M. Shand and S. Bryant, "A Framework for Loop-Free Convergence," IETF, RFC 5715, January 2010.

[17] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure, "Lossless Migrations of Link-State IGPs," *IEEE/ACM Trans. Netw.*, vol. 20, no. 6, pp. 1842–1855, December 2012.

[18] A. Shaikh, R. Dube, and A. Varma, "Avoiding Instability During Graceful Shutdown of Multiple OSPF Routers," *IEEE/ACM Trans. Netw.*, vol. 14, no. 3, pp. 532–542, June 2006.

[19] S. Vissicchio, L. Vanbever, L. Cittadini, G. Xie, and O. Bonaventure, "Safe Routing Reconfigurations with Route Redistribution," in *INFOCOM*, 2014.

[20] L. Vanbever, S. Vissicchio, L. Cittadini, and O. Bonaventure, "When the cure is worse than the disease: the impact of graceful igp operations on bgp," in *INFOCOM*, 2013.

APPENDIX

A. Algorithms

We now provide more information on GBA internal procedures reported in Algorithm 4.

Function **INIT** initializes an *mdag* structure by computing, for the affected source nodes in *SRC*, the merging of the two RSPDAGs and the list of Δ values. This function also sets the first *swallowing list* with all nodes in *SRC* that have either no successors or no predecessors in *mdag*.

Function **SWALLOW** iterates over nodes in the prepared *swallowing list*, removing from the *mdag* each node in the list and, recursively, all their neighbors that have either no successors or no predecessors when removing them from the current *mdag*. In addition, the function maintains a variable with the largest Δ value among the removed nodes (l. 11) and, if necessary[2], a list of *roots* containing, for each removed node, its predecessors in the initial routing graph that are still in *mdag* (l. 12). The root list allows for efficiently exploring the remaining graph.

Function **POPMAX** starts by permanently pruning from the *mdag* all nodes that are not, or no longer, involved in any transient loop (l. 2). Then, this function *clones* this current state of the *mdag* (l. 3) and repeatedly performs *swallowing* operations on a temporary copy in order to extract the new last constraint. At each iteration, the *minimum* Δ *value* among the remaining nodes in the copy is extracted (l. 5) and the associated nodes are added to the next *swallowing list* (l. 6–8). It iterates this way until no cycles appear. Eventually, the minimum Δ value computed at the last step is stored as the next *last constraint* to be considered by GBA (l. 12).

Function **UPMAX** initializes the next *swallowing list* with each node in *roots* whose associated Δ value is greater than m (l. 4–6). Those nodes depitcs routers that have already satisfied loop constraints (if any). Besides, the predecessors in the initial routing graph (**prePred**) of each node to be *swallowed* are added to the *root list* (l. 7). **UPMAX** iterates this way until all its elements have been treated.

[2]Note that the modifications on this *root list* have an actual impact only when performing a **SWALLOW** operation on the original *mdag*, and lines 12–13 could be skipped on the copy.

Algorithm 4 GBA internal procedures

```
1: function DAG.INIT (dag, dag', SRC, n)
2:     mdag.nodes = merge (dag, dag', SRC)
3:     for u in SRC do
4:         mdag.Δ[u] = C'(u,d) − C(u,d)
5:         if mdag.succ(u) = ∅ or mdag.pred(u) = ∅ then
6:             list.append (u)
7:     mdag.roots = {n}
8:     return mdag, list

1: function MDAG.SWALLOW (list)
2:     for LF in list do
3:         for s in succ(LF) do
4:             pred(s).remove (LF)
5:             if pred(s) = ∅ then
6:                 list.append (s)
7:         for p in pred(LF) do
8:             succ(p).remove (LF)
9:             if succ (p) = ∅ then
10:                list.append (p)
11:        max = max (max, Δ[LF])
12:        roots.append (prePred(LF))
13:        roots.remove (LF)
14:        remove (LF)

1: function MDAG.POPMAX (list)
2:     SWALLOW (list)
3:     GF = graphClone()
4:     while GF.nodes ≠ ∅ do
5:         GF.lc = GF.minΔ ()
6:         for n in GF.nodes do
7:             if Δ[n] == GF.lc then
8:                 min_ns.append (n)
9:         GF.lc = GF.lc + 1
10:        GF.SWALLOW (min_ns)
11:    max = GF.max
12:    lc = GF.lc

1: function MDAG.UPMAX (m)
2:     list = ∅
3:     for r in roots do
4:         if Δ[r] > m then
5:             list.append (r)
6:             roots.remove (r)
7:             roots.append (prePred(r))
8:     POPMAX (list)
```

B. Problem Statement Properties

The following properties hold by definition of Δ vectors and loop constraints. We refer to a generic loop L consdering a given destination d, its corresponding constraint c, and a given weight increment v applied to links outgoing from 0. For any router x, we denote its successors in $RSPDAG(d, G)$ and $RSPDAG(d, G')$ as its initial and final next-hops to d, respectively.

Property A.1. *Given a constraint* c, $\forall i \in \{1, \ldots, |c|\}$, $\bar{c}[i] \geq \underline{c}[i] + 2$.

Property A.2. *If* v *meets* c*, there exist at least two routers* $x, z \in L$ *such that (i)* x *uses its final next-hops* $y_1, \ldots, y_n$ *to* d*, with* $y_1, \ldots, y_n \notin L$ *because* $\Delta_d(x) = \underline{c}$*; and (ii)* z *uses its initial next-hops* $w_1, \ldots, w_n$ *to* d*, with* $w_1, \ldots, w_n \notin L$ *because* $\Delta_d(z) = \bar{c}$.

Property A.3. *All routers $x \in L$ use their respective (i) initial next-hops to d if v precedes c, and (ii) their final next-hops to d if v follows c.*

The following theorems about safety of a sequence follows from the properties above.

Theorem 1. *A weight sequence s avoids a loop L if and only if s contains only safe transitions with respect to the constraint corresponding to L.*

Proof: Let $c = (\underline{c}, \bar{c})$ and d respectively be the loop constraint and the destination associated to loop L. We prove the statement in two steps.

- if s includes an unsafe transition $(v_i\ v_{i+1})$ for c, then s does not prevent L. Indeed, by definition of unsafe transition, we have two cases: (i) v_i precedes $\underline{c}$ and v_{i+1} follows $\bar{c}$, and (ii) v_i follows $\underline{c}$ and v_{i+1} precedes $\bar{c}$. All the routers in L will switch from their initial to their final next-hops to d in the first case, and from their final to their initial next-hops in the second case. In both cases, the transition from v_i to v_{i+1} can cause L to occur by definition of transient loop.
- if s only includes safe transitions for c, then s prevents L. Indeed, by definition of safe transition, for each pair of weight increments v_i and v_j, where v_i precedes c, v_j follows c and $j > i$, there must exist a vector v_k such that $i < k < j$ and v_k meets c. By Property A.2, this means that each time routers in L switch from their initial to their final next-hops, there is an intermediate step (corresponding to v_k) in which some routers switch before others in such a way that the possible loop is prevented. A symmetric argument can be applied to the case in which v_i follows c and v_j precedes c.

The two cases prove the statement. ■

Theorem 2. *An always increasing weight sequence s avoids a loop L if and only if s contains at least one vector meeting the constraint corresponding to L.*

Proof: Let $c = (\underline{c}, \bar{c})$ be the constraint corresponding to any loop L. By definition of always increasing sequence, s is a concatenation of three subsequences, $s = l\ m\ h$, where l is composed by vectors preceding $\underline{c}$, m contains vectors meeting c, and h includes vectors following $\bar{c}$. By hypothesis, m cannot be empty. Thus, s does not contain unsafe transitions for c. The statement then follows by Theorem 1. ■

Also, the following theorem follows by the definition of intermediate forwarding CPC (see Section IV).

Theorem 3. *If a weight increment v satisfies the CPCs for all destinations, then no forwarding change occurs.*

Proof: Assume by contradiction that a forwarding change occurs for a destination d when v is applied, even if v verifies all the CPCs for d. By definition of forwarding change, a node $\bar{x}$ must exist such that one of its shortest paths to d after the application of v is not included either in the initial nor in the final ones. Since only the weights of the links outgoing from 0 are changed by v, the paths from $\bar{x}$ to 0 are the same as the initial ones. This means that 0 must also have changed its shortest paths to d.

By definition of CPCs, all the paths from 0 to d via initial successors have the same length after the application of v. Thus, for a forwarding change on 0 to occur, it must exist a path $(0\ x \dots d)$ shorter or equal than the shortest paths from 0 to d via any initial successor s^*. Since only the weights of the links outgoing from 0 are changed by v, this means that $v[s^*] + C(0,d) \geq v[x] + C(0,x,d)$, i.e., $v[s^*] \geq v[x] - offset[d][x]$. This inequality contradicts the hypothesis that all CPCs are verified by v, thus proving the statement. ■

C. Safety and minimality proofs

We now prove correctness and optimality of the AGBA algorithm with respect to the MCLP problem (see Section IV). Since MLP is a sub-problem of MCLP and GBA coincides with AGBA when CPCs are not enforced, the following proofs also show correctness and optimality of GBA with respect to the MLP problem (see Section III), as already proved in [12].

In our proofs, we use the term *before* iteration j to denote all iterations that are lower than j. We also say that a constraint is *unsatisfied* (resp., *satisfied*) at an iteration j if it is not met (resp., it is met) by any vector computed by AGBA before j. Finally, given an increment vector v and a set of CPCs, we recall that the *pivot component* is the biggest component of v appearing in the left side of any CPC inequality, i.e., a component corresponding to an initial successor (see Section IV). For simplicity, we restrict to the case of a single pivot component per vector. However, lemmas and theorems can be easily generalized to multiple pivot components.

Our proofs leverage the following properties of AGBA which hold by definition of the algorithm.

Property A.4. *At each iteration j,* AGBA *computes a vector v such that $v >^+ \underline{c}$ for all the constraints $c = (\underline{c}, \bar{c})$ still unsatisfied before j.*

Property A.5. *Given any* AGBA *iteration j, let v be the vector that* AGBA *computes before the adjusting phase. For each component i of v, a constraint $c = (\underline{c}, \bar{c})$ still unsatisfied before j exists such that $v[i] = \max(\underline{c}[i] + 1, 0)$.*

Property A.6. AGBA *computes always increasing sequences.*

Property A.7. AGBA *stops as soon as all the constraints are met. Each constraint is met by one vector in the sequence.*

Properties A.4 and A.5 are ensured by the greedy vector computation (plus the fact that AGBA only increases some components of the vector during the adjusting phase). Property A.6 is the result of both vector computation and constraint removal. Properties A.7 derives from the constraint removal mechanism.

First of all, we show that AGBA always terminates.

Lemma 1. *For any* AGBA *iteration, the pivot component of the computed vector remains bigger than any component appearing in the left side of any CPC inequality during the adjusting phase.*

Proof: The statement hold at the beginning of the adjusting phase by definition of pivot component.

Now, assume by contradiction that the statement holds until a given step s during the adjusting phase, but not after s. That is, at step s AGBA computes a vector in which at least one component m is bigger than the pivot component j, and m appears in the left side of some CPC inequalities. Let w and z be the vectors computed by AGBA respectively before and after step s. By hypothesis, $\forall i\ w[i] \leq w[j]$ while $z[m] > z[j]$.

This hypothesis implies that AGBA has increased the m-th component of w at step s. By definition, AGBA increases a component only if it appears in the right side of an CPC inequality. Thus, the inequality considered by AGBA in s must be $v[l] \leq v[m] + k$, with $k \geq 0$ and $l \neq m$. To accommodate this inequality, by definition, AGBA enforces $z[l] = z[m] + k$, that is, $z[l] \geq z[m]$ since $k \geq 0$. All the other components are left unmodified, hence $z[l] = w[l]$ and $z[j] = w[j]$.

We have two cases. If $l = j$, then $z[l] \geq z[m]$ implies $z[j] \geq z[m]$, which contradicts the definition of z. Otherwise, if $l \neq j$, then it must be $w[l] = z[l] \geq z[m] > z[j] = w[j]$, i.e., $w[l] > w[j]$, which contradicts the definition of w. In both cases, we contradict the hypothesis, which yields the statement. ■

Lemma 2. *The pivot component is never modified by AGBA during the adjusting phase.*

Proof: Let p be any vector before the adjusting phase, and let $p[j]$ be its pivot component. We now show that AGBA never modifies $p[j]$.

In the adjusting phase, AGBA iterates once on the sorted set of CPC inequalities, considering one inequality at the time and increasing some components of the vector if needed. Consider any step s in this iteration. Let w be the vector at the beginning of s. The following cases apply to the CPC inequality that AGBA considers at s.

- $v[j]$ does not appear in the inequality, hence it is not modified, by definition of AGBA.
- $v[j]$ appear in the left side of the inequality, which has the form $v[j] \leq v[i] + k$, with $i \neq j$ and $k \geq 0$. If the inequality is satisfied, AGBA will not modify any component of w. Otherwise, by definition, AGBA will only increase the value of $w[i]$ while not modifying $w[j]$.
- $v[j]$ appear in the left side of the inequality, which has the form $v[l] \leq v[j] + q$, with $l \neq j$ and $q \geq 0$. By Lemma 1, it must be $w[l] < w[j]$. Hence, the inequality is already satisfied by w, and by definition, AGBA does not modify any component of w.

In all the cases, AGBA does not modify $w[j]$. The statement follows by applying the same argument to all the steps performed by AGBA during the adjusting phase. ■

Lemma 2 implies that at least one constraint is satisfied by AGBA at each step. Indeed, the following lemma holds.

Lemma 3. *At each iteration,* AGBA *computes a vector v that meets at least one constraint not met before.*

Proof: Consider any AGBA iteration i. Let $\mathcal{C}$ be the set of unsatisfied constraints at the beginning of i, let v be the computed vector before the adjusting phase, and let $v[j]$ be its pivot component. By Property A.4 and A.5, one constraint $c = (\underline{c}, \bar{c}) \in \mathcal{C}$ must exist such that $v >^{+} \underline{c}$ and $v[j] = \underline{c}[j]+1$. By Property A.1, it must also be $v[j] < \bar{c}[j]$, that is, c is met by v. The statement follows by noting that $v[j]$ is unmodified by AGBA in the adjusting phase, by Lemma 2. ■

We now leverage Lemma 3 to prove that AGBA always terminates in a finite number of iterations.

Theorem 4. *For any MCLP instance $I =< \mathcal{C}, \mathcal{A} >$,* AGBA *always terminates in $O(|\mathcal{C}|)$ iterations.*

Proof: By Property A.7, AGBA stops when all the initial constraints $\mathcal{C}$ are met. Hence, the statement directly follows by Lemma 3. ■

We now show that the sequences computed by AGBA are guaranteed to be safe and to avoid intermediate edges.

Lemma 4. *In AGBA, adjusting a vector according to a given CPC does not invalidate previously satisfied CPCs.*

Proof: Assume by contradiction that AGBA invalidates a previously satisfied CPC inequality (1) $v[l] \leq v[m] + k$ to satisfy another CPC inequality (2) $v[i] \leq v[j]+q$. By definition of CPC, $k, q \geq 0$. Let w and z be the vectors computed during the adjusting phase immediately before and immediately after considering (2), respectively. Our assumption translates to having $w[l] \leq w[m] + k$, $z[i] \leq z[j] + q$, and $z[l] > z[m] + k$. One of the following cases must hold.

- w is already compliant with (2). Then, by definition, AGBA does not modify any component of the current vector w, hence $z[l] = w[l]$ and $z[m] = w[m]$. By definition of w, this means that it must be $z[l] \leq z[m]+k$, which contradicts the assumption.
- w is not compliant with (2) and $j \neq l$. By definition, AGBA only increases the j-th component of w, i.e., $z[j] > w[j]$ but $z[l] = w[l]$ and $z[m] = w[m]$. By definition of w, this implies that $z[l] \leq z[m] + k$, which contradicts the assumption.
- w is not compliant with (2) and $j = l$. Since (1) has been considered by AGBA before (2), then it must be $w[l] = w[j] > w[i]$ by definition of the sorting phase in AGBA. This means that (2) has been already satisfied by w, contradicting the hypothesis of this case.

All cases lead to a contradiction, yielding the statement. ■

Theorem 5. *The weight sequences computed by* AGBA *prevent both transient loops and intermediate edges.*

Proof: Let I be any MCLP instance, and let s be the sequence computed by AGBA on I. By Property A.6, s is an always increasing sequence. By Lemma 3, each constraint is met by at least one vector in s. Thus, Theorem 2 ensures the prevention of transient loops. Moreover, by definition of the AGBA adjusting phase and by Lemma 4, all the CPCs inequalities are satisfied by each weight increment in s. Hence, Theorem 3 guarantees the prevention of intermediate edges. ■

Finally, we prove the minimality of the sequences computed by AGBA.

Lemma 5. *Let I be any MCLP instance, $s = (s_1 \dots s_n)$ be any sequence solving I, and $g = (g_1 \dots g_m)$ be the sequence computed by* AGBA *on I, with possibly $n \neq m$. The last vectors of the sequences verify $v_n \geq g_m$.*

Proof: Let $I =< \mathcal{C}, \mathcal{A} >$. Assume by contradiction that $s_n[i] < g_m[i]$ for a given component i. We have two cases.

- *AGBA did not modify the i-th component in the adjusting phase* of its first iteration. Then, by Property A.5, there must exist at least one constraint $c = (\underline{c}, \bar{c}) \in \mathcal{C}$ such that $g_m[i] = \underline{c}[i] + 1$. This implies that $s_n[i] < \underline{c}[i] + 1$, hence c is not met by s_n.
- *AGBA modified the i-th component in the adjusting phase* of its first iteration. Then, by definition of adjusting phase, $\mathcal{A}$ must include an CPC inequality $v[j] \leq v[i]+y$. Since i-th component was actually adjusted by hypothesis, it must be $y > 0$, and AGBA enforced $g_m[j] = g_m[i] + y$, i.e., $g_m[j] < g_m[i]$. Moreover, for s to prevent intermediate edges, $s_n[j] \leq s_n[i] + y$. Since $s_n[i] < g_m[i]$ by hypothesis, it must be $s_n[j] < g_m[i] + y$, hence $s_n[j] < g_m[j]$.

In the second case, we can iterate the argument above starting from j-th component. Each time the second case applies, we end up with a component of g_m strictly bigger than the previously considered one. Thus, the second case can hold until we reach the biggest component of g_m that appears in the left side of any CPC inequality. By Lemma 1, this component is the pivot component. Thus, Lemma 2 ensures that the first case eventually applies.

Hence, at least one constraint c is not met by s_n. This means that s contains an unsafe transition for c, since s_n is the last weight increment in s and the final weight assignment is greater or equal than $\bar{c}$. Theorem 1 implies that s does not solve I, contradicting the hypothesis and yielding the statement. ■

Lemma 6. *Let I be any MCLP instance, $s = (s_1 \dots s_n)$ be any sequence solving I, and $g = (g_1 \dots g_m)$ be the sequence computed by* AGBA *on I, with possibly $n \neq m$. All the loop constraints met by s_n (and possibly more) are also met by g_m.*

Proof: Let $I =< \mathcal{C}, \mathcal{A} >$. Assume by contradiction that a constraint c is met by s_n, but not by g_m. By Property A.4, $g_m >^+ \underline{c}$. Hence, for g_m not to meet c, it must be $g_m >^+ \bar{c}$. Also, in order for s_n to meet c, it must exist a component i such that $s_n[i] < \bar{c}[i]$. As a result, $s_n[i] < \bar{c}[i] < g_m[i]$. This contradicts Lemma 5, hence yielding the statement. ■

Theorem 6. AGBA *computes a minimal sequence solving any given MCLP instance.*

Proof: Consider an MCLP instance $I =< \mathcal{C}, \mathcal{A} >$. Let $s^* = (s_1^* \dots s_n^*)$ and $g = (g_1 \dots g_m)$, with $n \leq m$, be respectively any minimal solution of I and the sequence computed by GBA on I. If $m = 1$, n must be equal to 1 as well, and the statement directly follows. Otherwise, we know by Lemma 6 that if g_m meets a set $\mathcal{C}_1 \subseteq \mathcal{C}$ of loop constraints, then s_n^* meets a subset of constraints in $\mathcal{C}_1$. Consider now the sequences $(s_1^* \dots s_{n-1}^*)$ and $(g_1 \dots g_{m-1})$. Again by Lemma 6, g_{m-1} meets at least the same set of constraints as s_{n-1}^*. This implies that the sequence $(g_{m-1}\ g_m)$ meets at least the same constraints met by $(s_{n-1}^*\ s_n^*)$. By iterating the same argument, we can show that $(g_{m-n+1} \dots g_m)$ meets at least the same set of constraints as $(s_1^* \dots s_n^*)$. Thus, by definition of s^*, $(g_{m-n+1} \dots g_m)$ meets all the constraints in $\mathcal{C}$. Also, Property A.7 ensures that AGBA stops at g_{m-n+1}. Hence, it must be $m - n = 0$ and $|g| = |s^*|$, yielding the statement. ■

Note that AGBA is minimal even if the MCLP instance allows relative weight modification in $\mathbb{Z}$ while GBA ensures minimality for the MLP only if globally relative weight modifications stays in $\mathbb{N}$.