Graceful Operations in Link-State Routing Networks

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

[1]UDS (France), [2]UCL (Belgium), [3]Cisco

June 12th, 2013

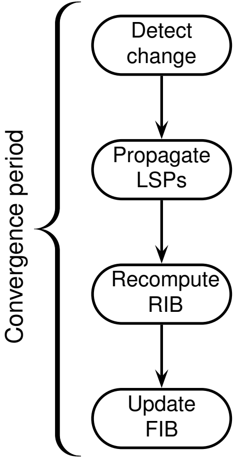
Research unit in networking seminar

Some context

- Routing in ISP networks (intra-domain)
  - Link-state protocols: OSPF, IS-IS
- Frequent topological changes
  - Maintenance operations on links or nodes
  - Traffic engineering (weight modifications)
- . . . and as many convergence periods
  - Transiently inconsistent state
  - Possible traffic disruption

Some context

- Routing in ISP networks (intra-domain)
  - Link-state protocols: OSPF, IS-IS
- Frequent topological changes
  - Maintenance operations on links or nodes
  - Traffic engineering (weight modifications)
- . . . and as many convergence periods
  - Transiently inconsistent state
  - Possible traffic disruption



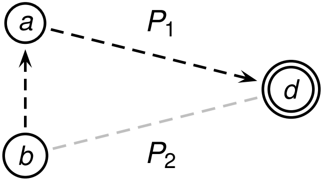
How do transient loops appear?

Routers' update order is **not controlled**!
(depends on *LSA flooding* and *RIB/FIB update* times)

Example:

- Initially, both *a* and *b* reach *d* through *a*;

Routes towards *d*:



$P_1 << P_2$

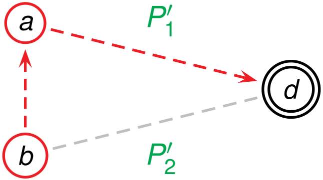
How do transient loops appear?

Routers' update order is **not controlled**!
(depends on *LSA flooding* and *RIB/FIB update* times)

Example:

- Initially, both *a* and *b* reach *d* through *a*;
- A change occur on the network;
  Path through b more interesting, even for a;

Routes towards *d*:



Old: $P_1 << P_2$
New: $P'_1 >> P'_2$

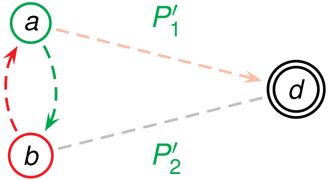
How do transient loops appear?

Routers' update order is **not controlled**!
(depends on *LSA flooding* and *RIB/FIB update* times)

Example:

- Initially, both *a* and *b* reach *d* through *a*;
- A change occur on the network;
  Path through b more interesting, even for a;
- If *a* updates first and starts sending data towards *d* through *b*, while *b* still uses *a*;

Routes towards *d*:



Old: $P_1 << P_2$
New: $P_1' >> P_2'$

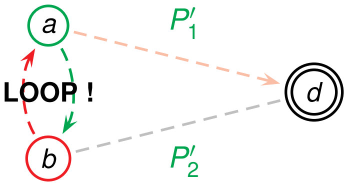
How do transient loops appear?

Routers' update order is **not controlled**!
(depends on *LSA flooding* and *RIB/FIB update* times)

Example:

- Initially, both *a* and *b* reach *d* through *a*;
- A change occur on the network;
  Path through b more interesting, even for a;
- If *a* updates first and starts sending data towards *d* through *b*, while *b* still uses *a*;
- A **transient loop** appears on link (a, b);

Routes towards d:



Old: $P_1 << P_2$
New: $P'_1 >> P'_2$

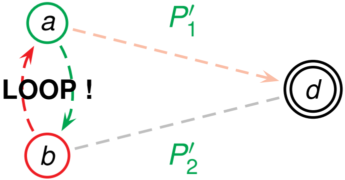
How do transient loops appear?

Routers' update order is **not controlled**!
(depends on *LSA flooding* and *RIB/FIB update* times)

Example:

- Initially, both a and b reach d through a;
- A change occur on the network;
  Path through b more interesting, even for a;
- If a updates first and starts sending data towards d through b, while b still uses a;
- A **transient loop** appears on link (a, b);
  - Increased latency;
  - Packet losses.

Routes towards d:



Old: $P_1 << P_2$
New: $P'_1 >> P'_2$

How to detect them?

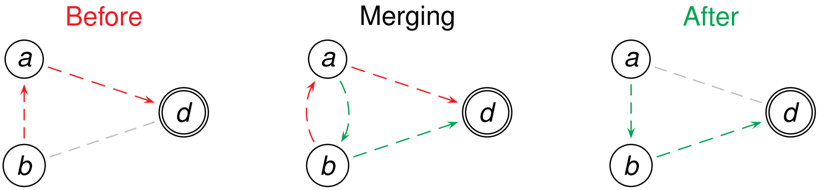
For a given destination (e.g. *d*):

1. Compute routes before and after the change;

How to detect them?

For a given destination (e.g. *d*):

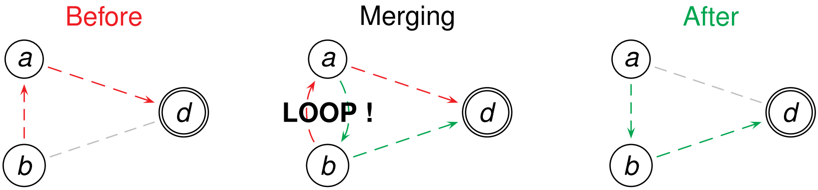
1. Compute routes before and after the change;
2. Merge these two directed acyclic graphs (DAG);



How to detect them?

For a given destination (e.g. *d*):

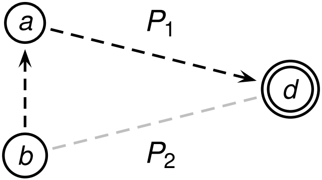
1. Compute routes before and after the change;
2. Merge these two directed acyclic graphs (DAG);
3. Perform a cycle detection on the resulting graph.



How to prevent them?

Force the routers to update in the *right* order.

- Initially, both *a* and *b* reach *d* through *a*;

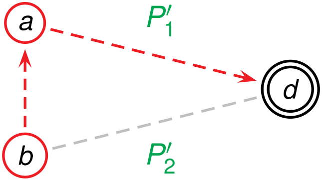


$$P_1 + w(b, a) < P_2$$

How to prevent them?

Force the routers to update in the *right* order.

- Initially, both *a* and *b* reach *d* through *a*;
- The same change occurs;

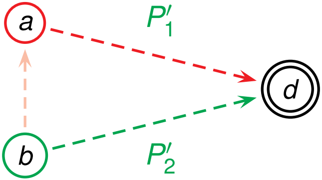


Old: $P_1 + w(b, a) < P_2$
New: $P'_1 > w(a, b) + P'_2$

How to prevent them?

Force the routers to update in the *right* order.

- Initially, both *a* and *b* reach *d* through *a*;
- The same change occurs;
- Yet this time *b* updates first;

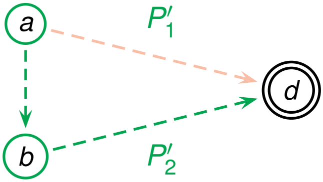


Old: $P_1 + w(b, a) < P_2$
New: $P'_1 > w(a, b) + P'_2$

How to prevent them?

Force the routers to update in the *right* order.

- Initially, both *a* and *b* reach *d* through *a*;
- The same change occurs;
- Yet this time *b* updates first;
- Then *a*, and no loop appears.

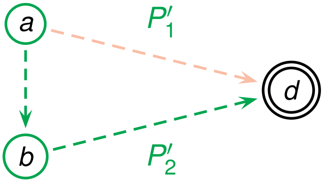


Old: $P_1 + w(b, a) < P_2$
New: $P'_1 > w(a, b) + P'_2$

How to prevent them?

Force the routers to update in the *right* order.

- Initially, both *a* and *b* reach *d* through *a*;
- The same change occurs;
- Yet this time *b* updates first;
- Then *a*, and no loop appears.



One goal, several approaches.

Old: $P_1 + w(b, a) < P_2$
New: $P'_1 > w(a, b) + P'_2$

Progressive update

Basic idea

Split up the change into a sequence **loop free** updates.

Objectives

Compute a sequence of intermediate updates, such that:

- No transient loop between subsequent updates;
- Each intermediate update prevents at least one cycle.

Challenge

Minimal operational impact (sequences of minimal length)

Illustration: path increment sequence

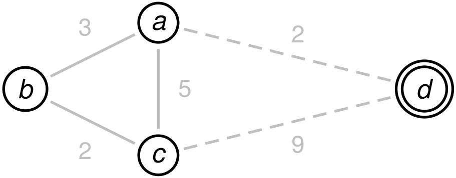


Illustration: path increment sequence

- Initially, *a*, *b* and *c* reach *d* through node *a*.

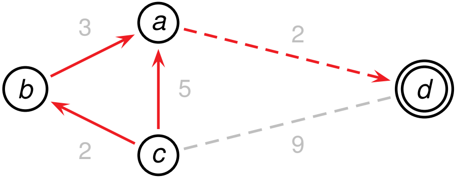


Illustration: path increment sequence

- Initially, *a*, *b* and *c* reach *d* through node *a*.
- If a change occur on path $P(a, d)$ increasing its cost to 50…

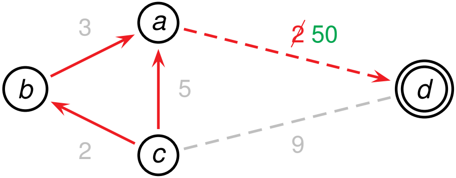


Illustration: path increment sequence

- Initially, *a*, *b* and *c* reach *d* through node *a*.
- If a change occur on path $P(a, d)$ increasing its cost to 50, all three nodes will go through *c* instead ...

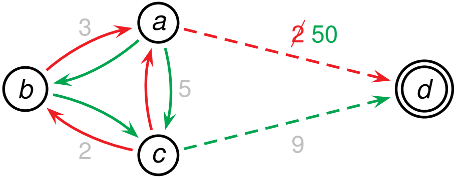


Illustration: path increment sequence

- Initially, *a*, *b* and *c* reach *d* through node *a*.
- If a change occur on path $P(a, d)$ increasing its cost to 50, all three nodes will go through *c* instead and transient loops may appear.

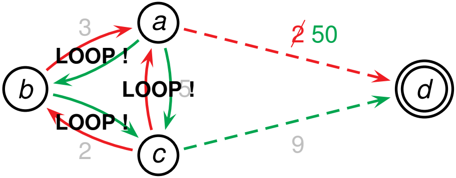


Illustration: path increment sequence

- Initially, *a*, *b* and *c* reach *d* through node *a*.
- If a change occur on path $P(a, d)$ increasing its cost to 50, all three nodes will go through *c* instead and transient loops may appear.

With incremental updates:

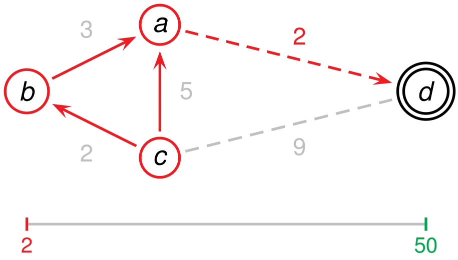


Illustration: path increment sequence

- Initially, *a*, *b* and *c* reach *d* through node *a*.
- If a change occur on path $P(a, d)$ increasing its cost to 50, all three nodes will go through *c* instead and transient loops may appear.

With incremental updates:

- Node *c* could update first;

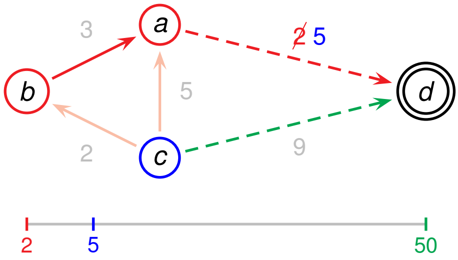


Illustration: path increment sequence

- Initially, *a*, *b* and *c* reach *d* through node *a*.
- If a change occur on path $P(a, d)$ increasing its cost to 50, all three nodes will go through *c* instead and transient loops may appear.

With incremental updates:

- Node *c* could update first;
- Then *b*,

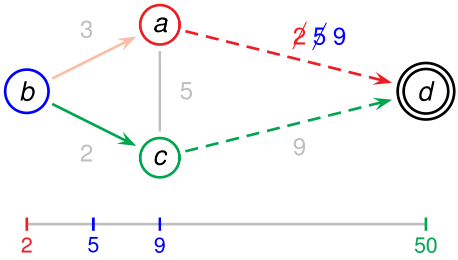


Illustration: path increment sequence

- Initially, *a*, *b* and *c* reach *d* through node *a*.
- If a change occur on path $P(a, d)$ increasing its cost to 50, all three nodes will go through *c* instead and transient loops may appear.

With incremental updates:

- Node *c* could update first;
- Then *b*, and *a*;

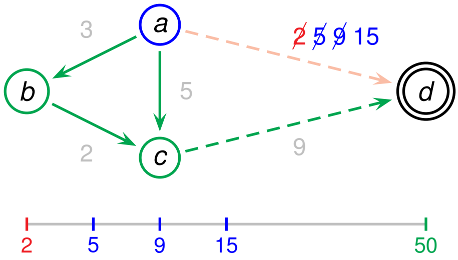


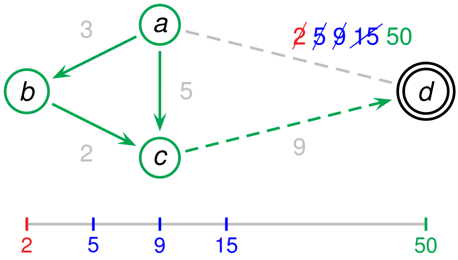
Illustration: path increment sequence

- Initially, *a*, *b* and *c* reach *d* through node *a*.
- If a change occur on path $P(a, d)$ increasing its cost to 50, all three nodes will go through *c* instead and transient loops may appear.

With incremental updates:

- Node *c* could update first;
- Then *b*, and *a*;

So that the transition to 50 will be loop free.



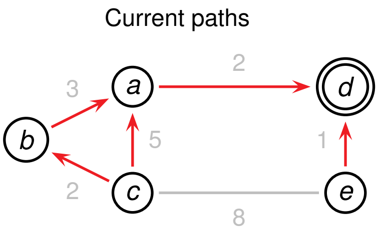
Case of a link shut (withdrawal)[1]

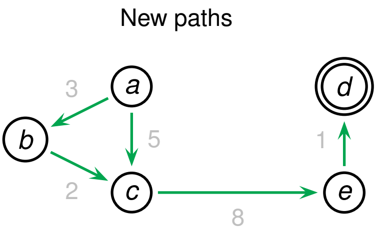
Algorithmic steps

1. Extract **destination oriented** increment sequences;
2. Merge them into a **global** increment sequence;
3. Prune useless values to build a **minimal** sequence.

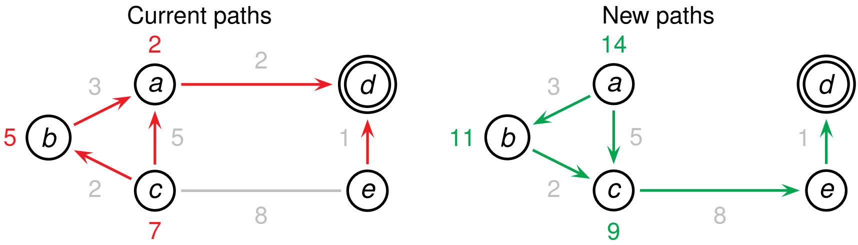
[1] The same algorithms may be used for any other kind of modification on a single link (addition, arbitrary weight increment or decrement).

Destination oriented sequences: Δ values



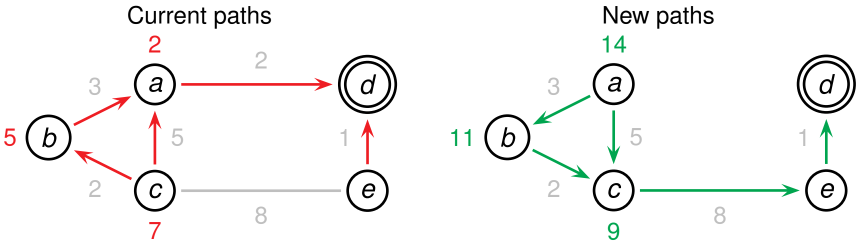


Destination oriented sequences: Δ values



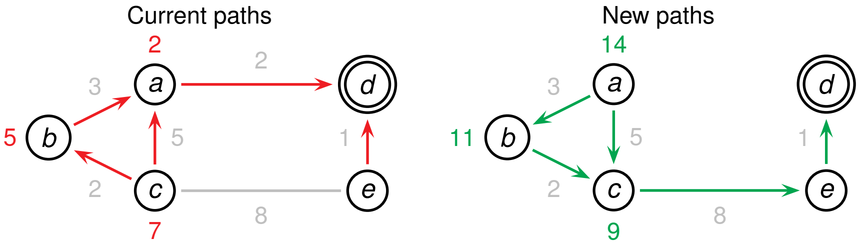
- Retrieve distances from each *affected* node to the destination

Destination oriented sequences: Δ values



- Retrieve distances from each *affected* node to the destination
- Compute the difference (Δ) between new and old distances
  - $\Delta(a) = 14 - 2 = 12$
  - $\Delta(b) = 11 - 5 = 6$
  - $\Delta(c) = 9 - 7 = 2$

Destination oriented sequences: Δ values

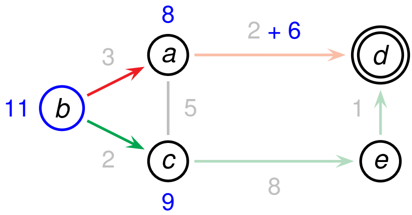


- Retrieve distances from each *affected* node to the destination
- Compute the difference (Δ) between new and old distances
  - $\Delta(a) = 14 - 2 = 12$
  - $\Delta(b) = 11 - 5 = 6$
  - $\Delta(c) = 9 - 7 = 2$

Incrementing the weight of link (a, d) by one of these Δ values would put the corresponding node in an **ECMP transient state.**

Destination oriented sequences: ECMP state

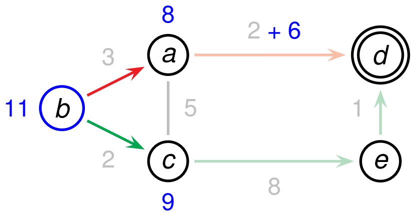
In an **ECMP state**, a node uses both its
old and new routes towards the destination.



- Δ sequence: $S_{\Delta}(d) = \{2, 6, 12\}$
  - ▷ First values such that the nodes use their new path(s)

Destination oriented sequences: ECMP state

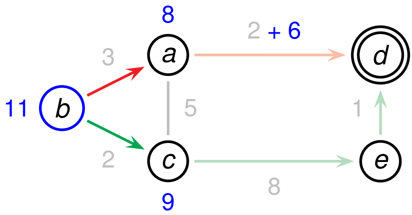
In an **ECMP state**, a node uses both its old and new routes towards the destination.



- Δ sequence: $S_{\Delta}(d) = \{2, 6, 12\}$
  - First values such that the nodes use their new path(s)
  - **Does not prevent transient loops**

Destination oriented sequences: ECMP state

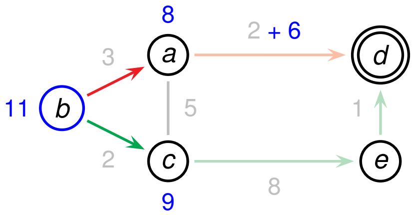
In an **ECMP state**, a node uses both its old and new routes towards the destination.



- Δ sequence: $S_\Delta(d) = \{2, 6, 12\}$
  - First values such that the nodes use their new path(s)
  - **Does not prevent transient loops**
- Increment seq. $(\Delta + 1)$: $S_i(d) = \{3, 7, 13\}$
  - First values such that the nodes use **only** their new path(s)

Destination oriented sequences: ECMP state

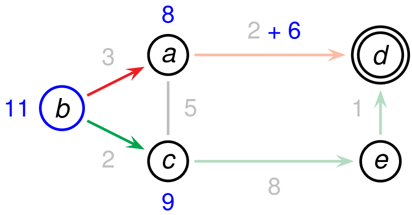
In an **ECMP state**, a node uses both its old and new routes towards the destination.



- Δ sequence: $S_\Delta(d) = \{2, 6, 12\}$
  - ▷ First values such that the nodes use their new path(s)
  - ▷ **Does not prevent transient loops**
- Increment seq. $(\Delta + 1)$: $S_i(d) = \{3, 7, 13\}$
  - ▷ First values such that the nodes use **only** their new path(s)
- Weight seq. $(\Delta + 1 + w(a, d))$: $S_m(d) = \{5, 9, 15\}$

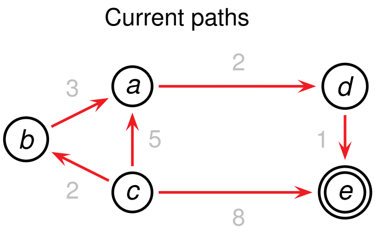
Destination oriented sequences: ECMP state

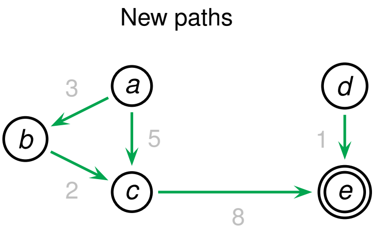
In an **ECMP state**, a node uses both its old and new routes towards the destination.



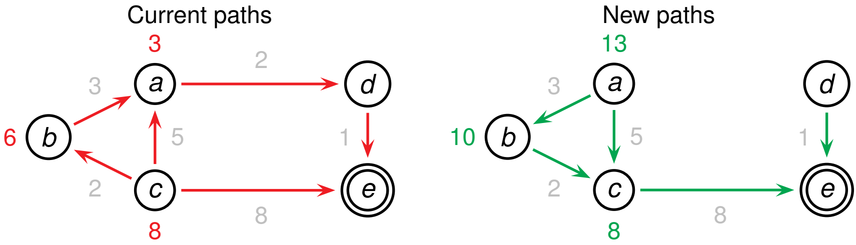
- Δ sequence: $S_\Delta(d) = \{2, 6, 12\}$
  - ▷ First values such that the nodes use their new path(s)
  - ▷ **Does not prevent transient loops**
- Increment seq. $(\Delta + 1)$: $S_i(d) = \{3, 7, 13\}$ relative to $w(a, d)$
  - ▷ First values such that the nodes use **only** their new path(s)
- Weight seq. $(\Delta + 1 + w(a, d))$: $S_m(d) = \{5, 9, 15\}$ absolute

Destination oriented sequences: another destination

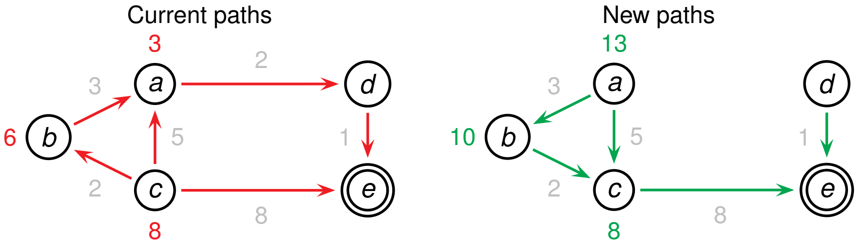




Destination oriented sequences: another destination

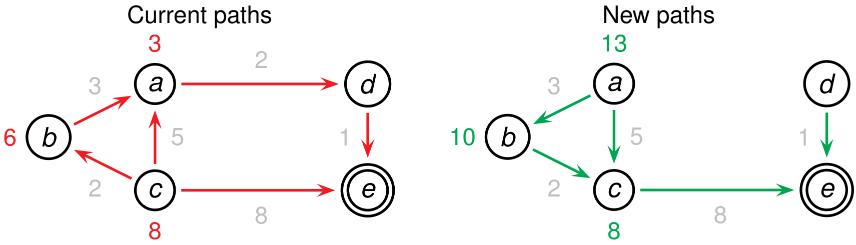


Destination oriented sequences: another destination



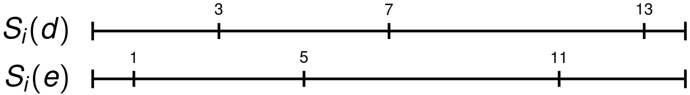
- Extract Δ values
  - $\Delta(a) = 13 - 3 = 10$
  - $\Delta(b) = 10 - 6 = 4$
  - $\Delta(c) = 8 - 8 = 0$

Destination oriented sequences: another destination

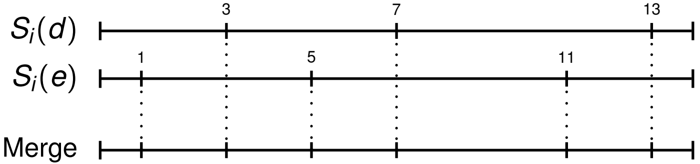


- Extract Δ values
  - $\Delta(a) = 13 - 3 = 10$
  - $\Delta(b) = 10 - 6 = 4$
  - $\Delta(c) = 8 - 8 = 0$
- Compute an increment sequence: $S_i(e) = \{1, 5, 11\}$

Global increment sequences

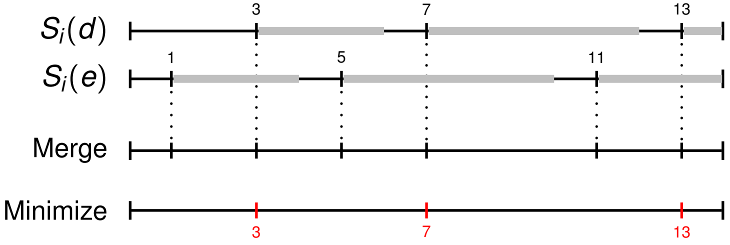


Global increment sequences



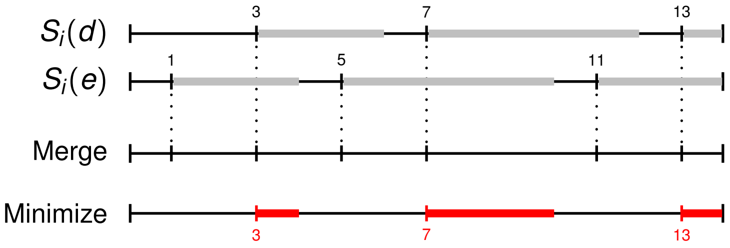
- Merge destination oriented sequences
  - Prevent transient loops for all destinations
  - May contain unnecessary values

Global increment sequences



- Merge destination oriented sequences
  - Prevent transient loops for all destinations
  - May contain unnecessary values
- Prune redundant values
  - Greedy algorithm looking for possible loops at each step
  - Ensure the minimality in terms of sequence length

Global increment sequences



- Merge destination oriented sequences
  - ▷ Prevent transient loops for all destinations
  - ▷ May contain unnecessary values
- Prune redundant values
  - ▷ Greedy algorithm looking for possible loops at each step
  - ▷ Ensure the minimality in terms of sequence length
  - ▷ Multiple sequences of minimal length

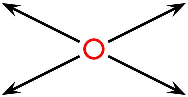
The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

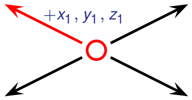
- Simple solution: shut down each link one by one
  - Number of intermediate steps proportional to node degree



The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

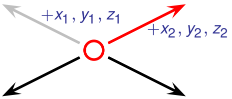
- Simple solution: shut down each link one by one
  - Number of intermediate steps proportional to node degree



The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

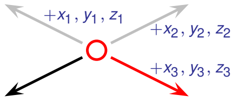
- Simple solution: shut down each link one by one
  - ▷ Number of intermediate steps proportional to node degree



The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

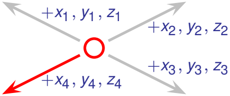
- Simple solution: shut down each link one by one
  - ▷ Number of intermediate steps proportional to node degree



The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

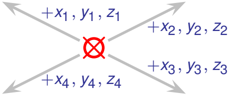
- Simple solution: shut down each link one by one
  - Number of intermediate steps proportional to node degree



The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

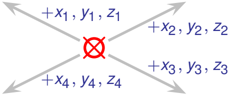
- Simple solution: shut down each link one by one
  - ▷ Number of intermediate steps proportional to node degree



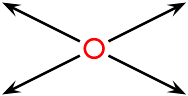
The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

- Simple solution: shut down each link one by one
  - ▷ Number of intermediate steps proportional to node degree



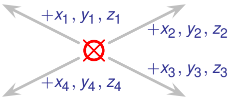
- Better solution: benefit from existing OSPF / IS-IS features
  - ▷ Simultaneous weight modifications



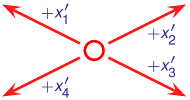
The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

- Simple solution: shut down each link one by one
  - ▷ Number of intermediate steps proportional to node degree



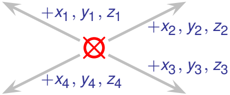
- Better solution: benefit from existing OSPF / IS-IS features
  - ▷ Simultaneous weight modifications



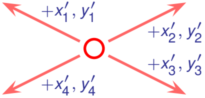
The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

- Simple solution: shut down each link one by one
  - ▷ Number of intermediate steps proportional to node degree



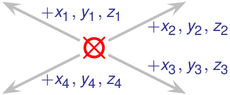
- Better solution: benefit from existing OSPF / IS-IS features
  - ▷ Simultaneous weight modifications



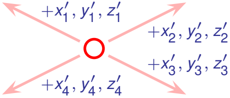
The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

- Simple solution: shut down each link one by one
  - ▷ Number of intermediate steps proportional to node degree



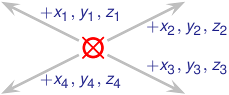
- Better solution: benefit from existing OSPF / IS-IS features
  - ▷ Simultaneous weight modifications



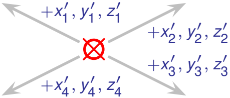
The Node Shutdown Problem

Objective: gracefully reroute the traffic out of a node.

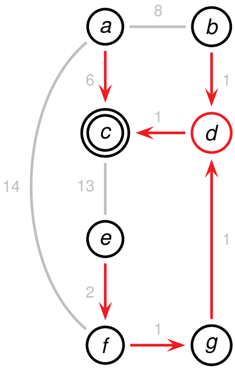
- Simple solution: shut down each link one by one
  - ▷ Number of intermediate steps proportional to node degree



- Better solution: benefit from existing OSPF / IS-IS features
  - ▷ Simultaneous weight modifications



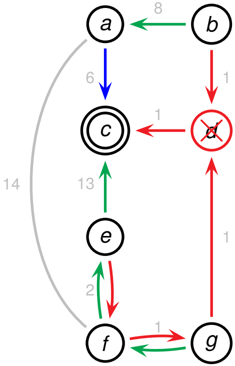
Towards Multi-Dimensional Increments



Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

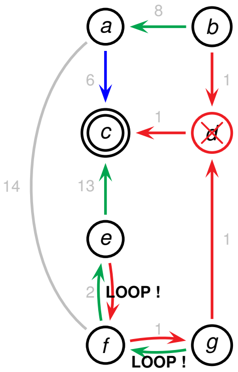
Towards Multi-Dimensional Increments



Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

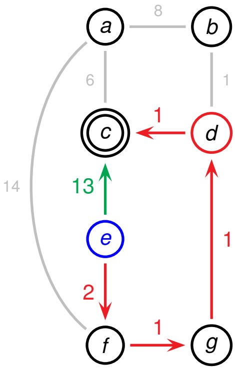
Towards Multi-Dimensional Increments



Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

Towards Multi-Dimensional Increments

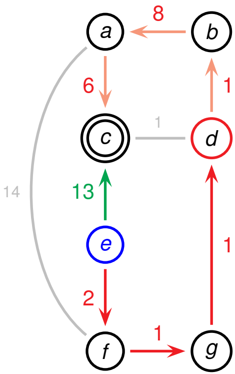


Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

- $\Delta_2^3(e) = \left(13 - (2+1+1+1) \right)$

Towards Multi-Dimensional Increments

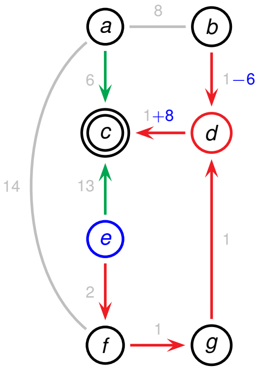


Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

- $\Delta_2^3(e) = \begin{pmatrix} 13 - (2 + 1 + 1 + 1) \\ 13 - (2 + 1 + 1 + 1 + 8 + 6) \end{pmatrix}$

Towards Multi-Dimensional Increments

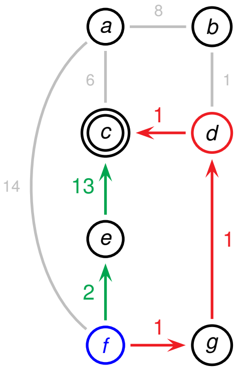


Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

- $\Delta_2^3(e) = \begin{pmatrix} 13-(2+1+1+1) \\ 13-(2+1+1+1+8+6) \end{pmatrix} = \begin{pmatrix} 8 \\ -6 \end{pmatrix}$

Towards Multi-Dimensional Increments

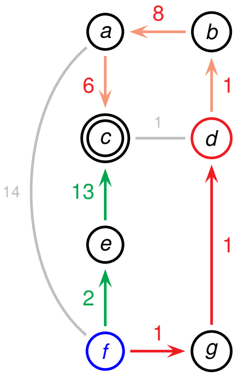


Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

- $\Delta_2^3(e) = \begin{pmatrix} 13 - (2+1+1+1) \\ 13 - (2+1+1+1+8+6) \end{pmatrix} = \begin{pmatrix} 8 \\ -6 \end{pmatrix}$
- $\Delta_2^3(f) = \begin{pmatrix} 15 - 3 \\ \end{pmatrix}$

Towards Multi-Dimensional Increments

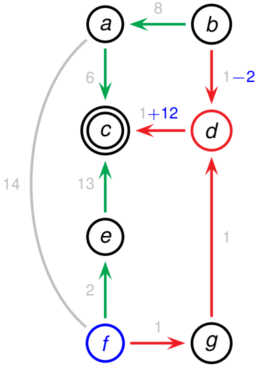


Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

- $\Delta_2^3(e) = \begin{pmatrix} 13-(2+1+1+1) \\ 13-(2+1+1+1+8+6) \end{pmatrix} = \begin{pmatrix} 8 \\ -6 \end{pmatrix}$
- $\Delta_2^3(f) = \begin{pmatrix} 15-3 \\ 15-17 \end{pmatrix}$

Towards Multi-Dimensional Increments

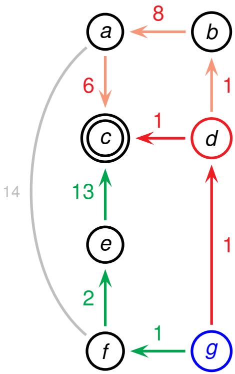


Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

- $\Delta_2^3(e) = \begin{pmatrix} 13-(2+1+1+1) \\ 13-(2+1+1+1+8+6) \end{pmatrix} = \begin{pmatrix} 8 \\ -6 \end{pmatrix}$
- $\Delta_2^3(f) = \begin{pmatrix} 15-3 \\ 15-17 \end{pmatrix} = \begin{pmatrix} 12 \\ -2 \end{pmatrix}$

Towards Multi-Dimensional Increments

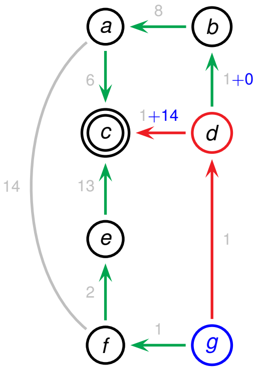


Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

- $\Delta_2^3(e) = \begin{pmatrix} 13-(2+1+1+1) \\ 13-(2+1+1+1+8+6) \end{pmatrix} = \begin{pmatrix} 8 \\ -6 \end{pmatrix}$
- $\Delta_2^3(f) = \begin{pmatrix} 15-3 \\ 15-17 \end{pmatrix} = \begin{pmatrix} 12 \\ -2 \end{pmatrix}$
- $\Delta_2^3(g) = \begin{pmatrix} 16-2 \\ 16-16 \end{pmatrix}$

Towards Multi-Dimensional Increments

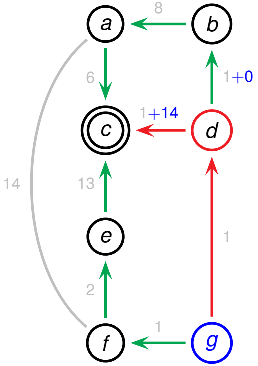


Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

- $\Delta_2^3(e) = \begin{pmatrix} 13-(2+1+1+1) \\ 13-(2+1+1+1+8+6) \end{pmatrix} = \begin{pmatrix} 8 \\ -6 \end{pmatrix}$
- $\Delta_2^3(f) = \begin{pmatrix} 15-3 \\ 15-17 \end{pmatrix} = \begin{pmatrix} 12 \\ -2 \end{pmatrix}$
- $\Delta_2^3(g) = \begin{pmatrix} 16-2 \\ 16-16 \end{pmatrix} = \begin{pmatrix} 14 \\ 0 \end{pmatrix}$

Towards Multi-Dimensional Increments



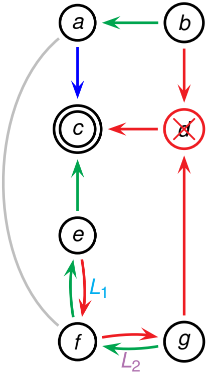
Vector of minimum increments such that a node x uses a new path, not through n, to reach d.

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

- $\Delta_2^3(e) = \begin{pmatrix} 13-(2+1+1+1) \\ 13-(2+1+1+1+8+6) \end{pmatrix} = \begin{pmatrix} 8 \\ -6 \end{pmatrix} \sim \begin{pmatrix} 8 \\ 0 \end{pmatrix}$
- $\Delta_2^3(f) = \begin{pmatrix} 15-3 \\ 15-17 \end{pmatrix} = \begin{pmatrix} 12 \\ -2 \end{pmatrix} \sim \begin{pmatrix} 12 \\ 0 \end{pmatrix}$
- $\Delta_2^3(g) = \begin{pmatrix} 16-2 \\ 16-16 \end{pmatrix} = \begin{pmatrix} 14 \\ 0 \end{pmatrix}$

Negative values denote the absence of constraint on a link.

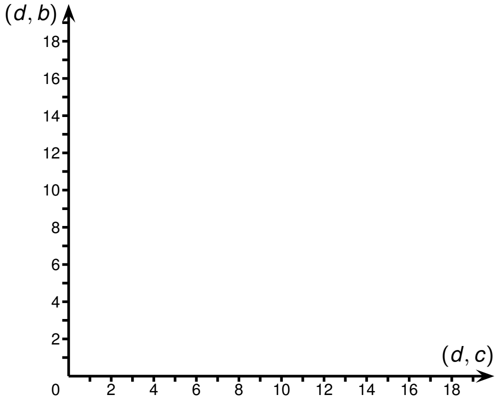
Modeling Loops as Vectorial Constraints



$$\Delta_2^3(e) = \begin{pmatrix} 8 \\ 0 \end{pmatrix}$$

$$\Delta_2^3(f) = \begin{pmatrix} 12 \\ 0 \end{pmatrix}$$

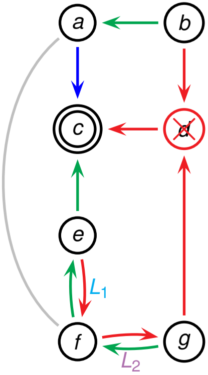
$$\Delta_2^3(g) = \begin{pmatrix} 14 \\ 0 \end{pmatrix}$$



Constraint c associated to a given a loop L.

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

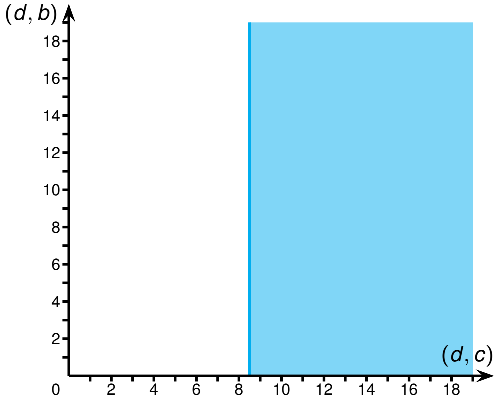
Modeling Loops as Vectorial Constraints



$$\Delta_2^3(e) = \begin{pmatrix} 8 \\ 0 \end{pmatrix}$$

$$\Delta_2^3(f) = \begin{pmatrix} 12 \\ 0 \end{pmatrix}$$

$$\Delta_2^3(g) = \begin{pmatrix} 14 \\ 0 \end{pmatrix}$$

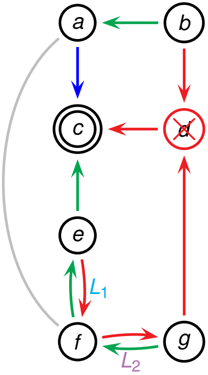


$$\underline{c}_1 = \begin{pmatrix} 8 \\ 0 \end{pmatrix}$$

Constraint c associated to a given a loop L.

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

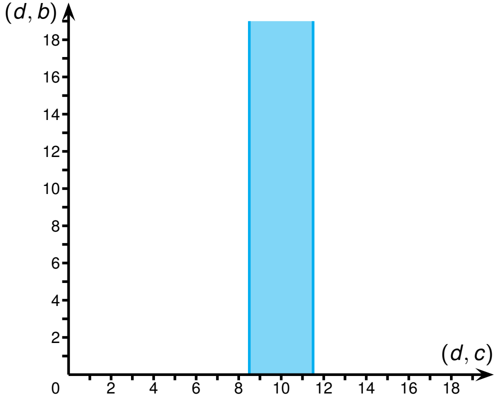
Modeling Loops as Vectorial Constraints



$$\Delta_2^3(e) = \begin{pmatrix} 8 \\ 0 \end{pmatrix}$$

$$\Delta_2^3(f) = \begin{pmatrix} 12 \\ 0 \end{pmatrix}$$

$$\Delta_2^3(g) = \begin{pmatrix} 14 \\ 0 \end{pmatrix}$$

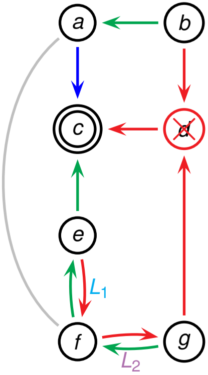


$$\underline{c}_1 = \begin{pmatrix} 8 \\ 0 \end{pmatrix}, \bar{c}_1 = \begin{pmatrix} 12 \\ 0 \end{pmatrix}$$

Constraint c associated to a given a loop L.

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

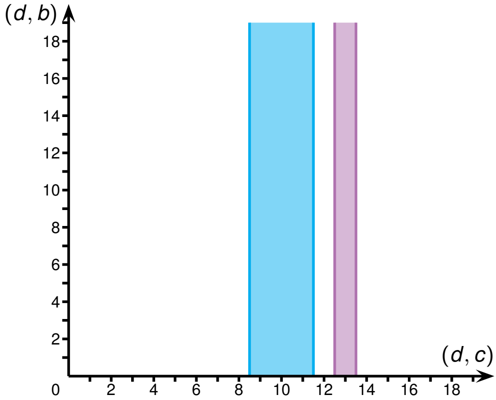
Modeling Loops as Vectorial Constraints



$$\Delta_2^3(e) = \begin{pmatrix} 8 \\ 0 \end{pmatrix}$$

$$\Delta_2^3(f) = \begin{pmatrix} 12 \\ 0 \end{pmatrix}$$

$$\Delta_2^3(g) = \begin{pmatrix} 14 \\ 0 \end{pmatrix}$$



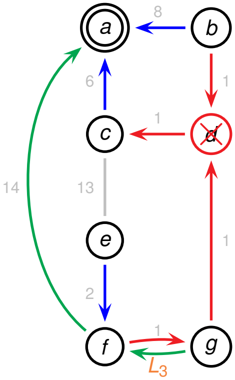
Constraint c associated to a given a loop L.

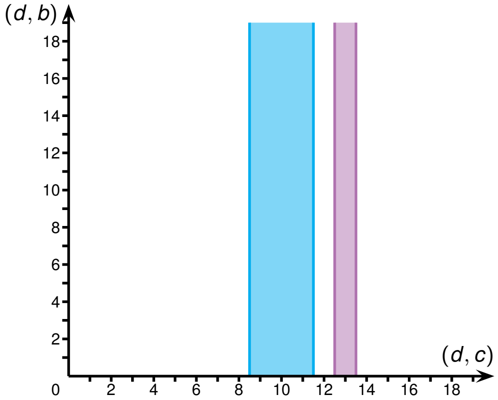
$$c := (\underline{c} := \min_{\forall x \in L}(\Delta(x)), \bar{c} := \max_{\forall x \in L}(\Delta(x)))$$

$$\underline{c}_1 = \begin{pmatrix} 8 \\ 0 \end{pmatrix}, \bar{c}_1 = \begin{pmatrix} 12 \\ 0 \end{pmatrix}$$

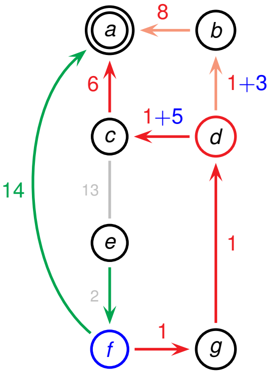
$$\underline{c}_2 = \begin{pmatrix} 12 \\ 0 \end{pmatrix}, \bar{c}_2 = \begin{pmatrix} 14 \\ 0 \end{pmatrix}$$

Modeling Loops as Vectorial Constraints (2)

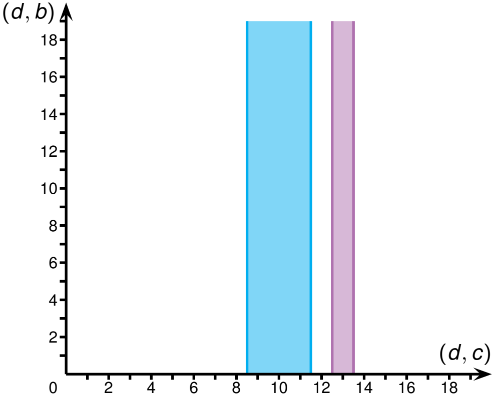




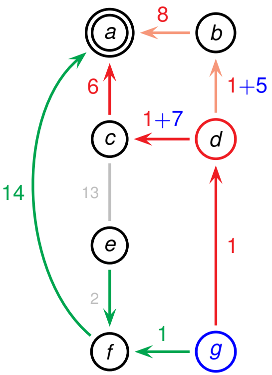
Modeling Loops as Vectorial Constraints (2)



$$\Delta_0^3(f) = \begin{pmatrix} 5 \\ 3 \end{pmatrix}$$

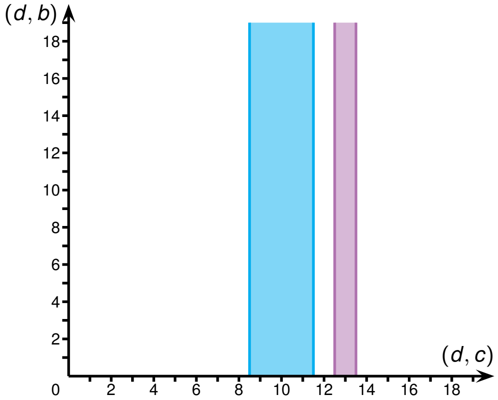


Modeling Loops as Vectorial Constraints (2)

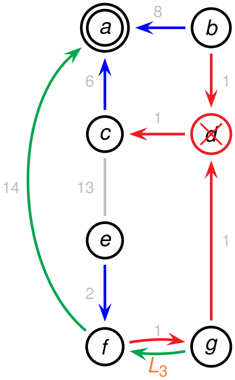


$$\Delta_0^3(f) = \begin{pmatrix} 5 \\ 3 \end{pmatrix}$$

$$\Delta_0^3(g) = \begin{pmatrix} 7 \\ 5 \end{pmatrix}$$

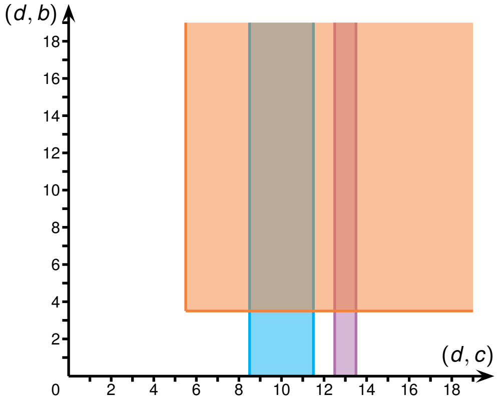


Modeling Loops as Vectorial Constraints (2)



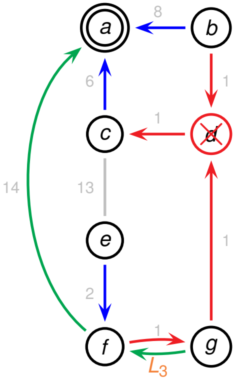
$$\Delta_0^3(f) = \begin{pmatrix} 5 \\ 3 \end{pmatrix}$$

$$\Delta_0^3(g) = \begin{pmatrix} 7 \\ 5 \end{pmatrix}$$



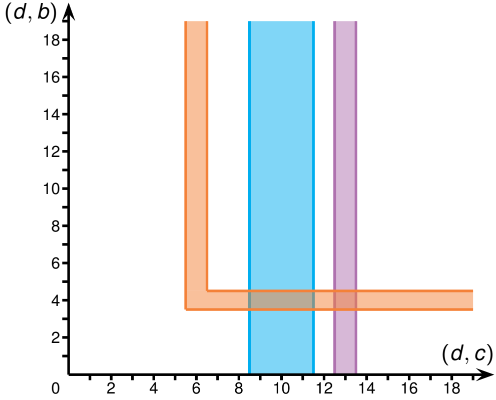
$$\underline{c_3} = \begin{pmatrix} 5 \\ 3 \end{pmatrix}$$

Modeling Loops as Vectorial Constraints (2)



$$\Delta_0^3(f) = \begin{pmatrix} 5 \\ 3 \end{pmatrix}$$

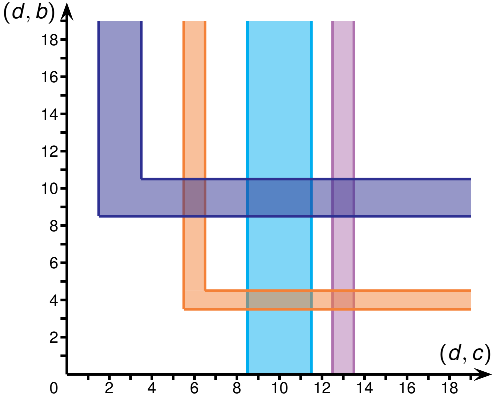
$$\Delta_0^3(g) = \begin{pmatrix} 7 \\ 5 \end{pmatrix}$$



$$\underline{c}_3 = \begin{pmatrix} 5 \\ 3 \end{pmatrix}, \quad \bar{c}_3 = \begin{pmatrix} 7 \\ 5 \end{pmatrix}$$

Modeling Loops as Vectorial Constraints (3)

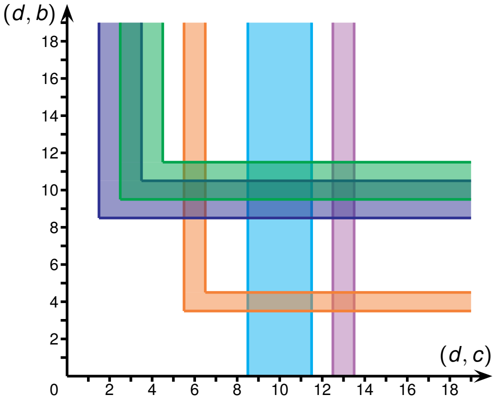
$$c_4 = \left(\begin{pmatrix} 1 \\ 8 \end{pmatrix}, \begin{pmatrix} 4 \\ 11 \end{pmatrix} \right)$$



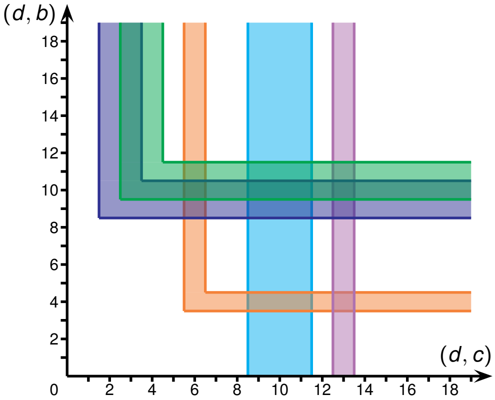
Modeling Loops as Vectorial Constraints (3)

$$c_4 = \left(\begin{pmatrix} 1 \\ 8 \end{pmatrix}, \begin{pmatrix} 4 \\ 11 \end{pmatrix} \right)$$

$$c_5 = \left(\begin{pmatrix} 2 \\ 9 \end{pmatrix}, \begin{pmatrix} 5 \\ 12 \end{pmatrix} \right)$$



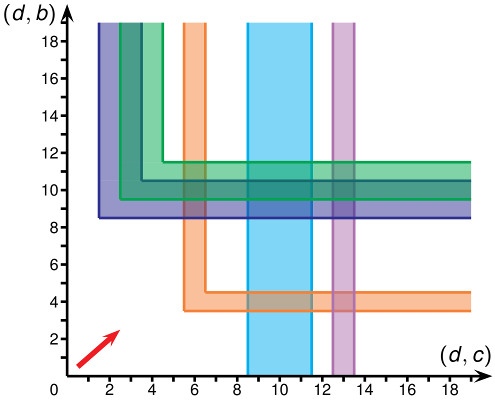
Defining Safe Weight Increment Sequences



A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Defining Safe Weight Increment Sequences

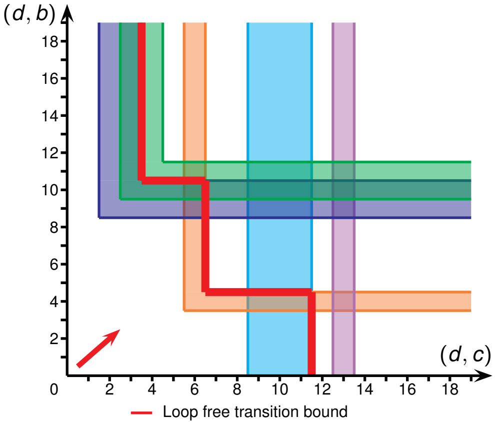
Forward search:



A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Defining Safe Weight Increment Sequences

Forward search:

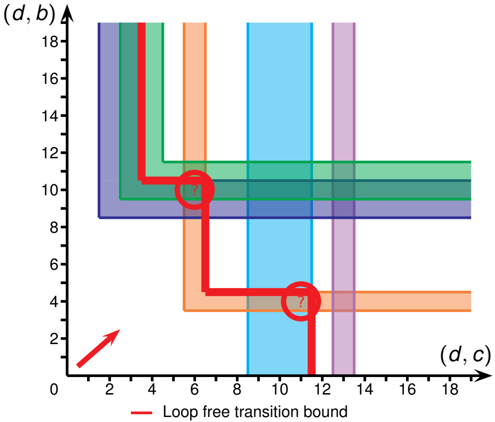


A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Defining Safe Weight Increment Sequences

Forward search:

1. c_3, c_4, c_5 || c_1, c_3 ?

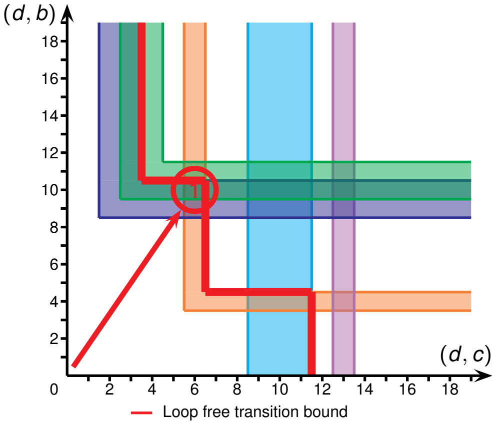


A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Defining Safe Weight Increment Sequences

Forward search:

1. c_3, c_4, c_5 *(greedy)*

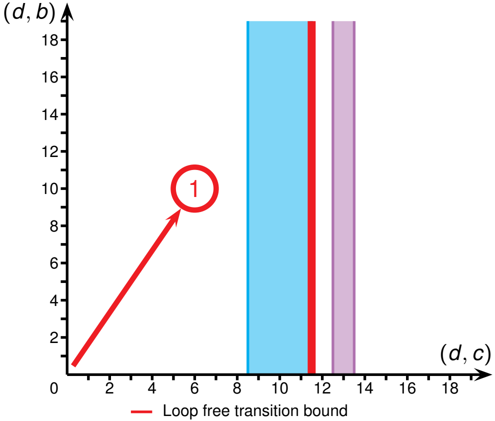


A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Defining Safe Weight Increment Sequences

Forward search:

1. c_3, c_4, c_5 *(greedy)*

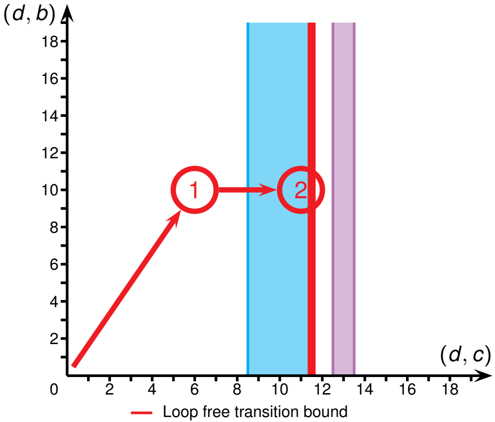


A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Defining Safe Weight Increment Sequences

Forward search:

1. c_3, c_4, c_5 *(greedy)*
2. c_1

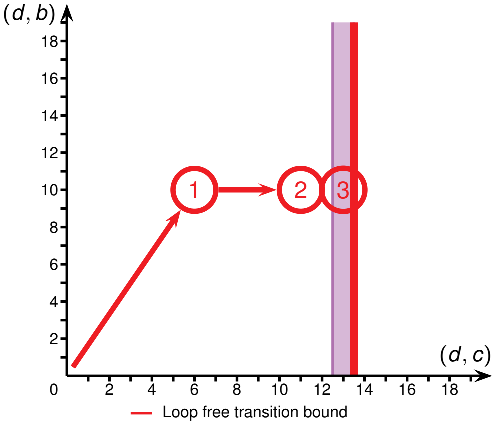


A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Defining Safe Weight Increment Sequences

Forward search:

1. c_3, c_4, c_5 *(greedy)*
2. c_1
3. c_2

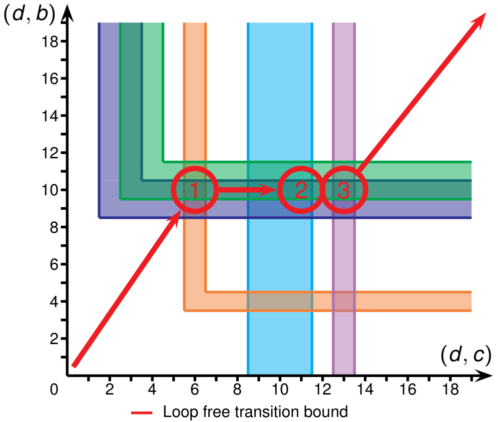


A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Defining Safe Weight Increment Sequences

Forward search:

1. c_3, c_4, c_5 *(greedy)*
2. c_1
3. c_2



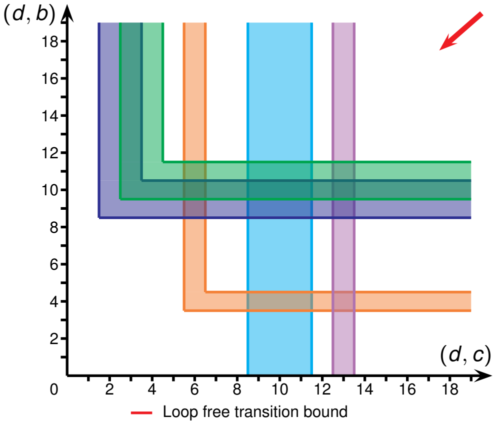
A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Defining Safe Weight Increment Sequences

Forward search:

1. c_3, c_4, c_5 *(greedy)*
2. c_1
3. c_2

Backward search:



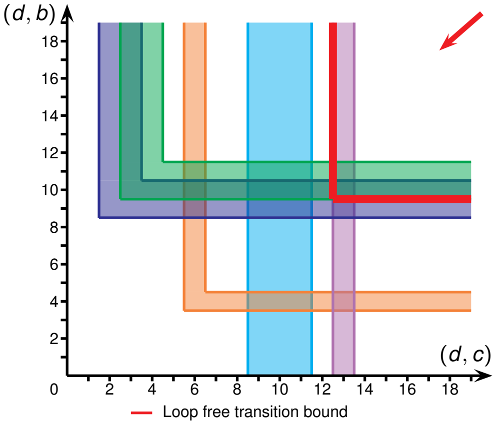
A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Defining Safe Weight Increment Sequences

Forward search:

1. c_3, c_4, c_5 *(greedy)*
2. c_1
3. c_2

Backward search:



A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

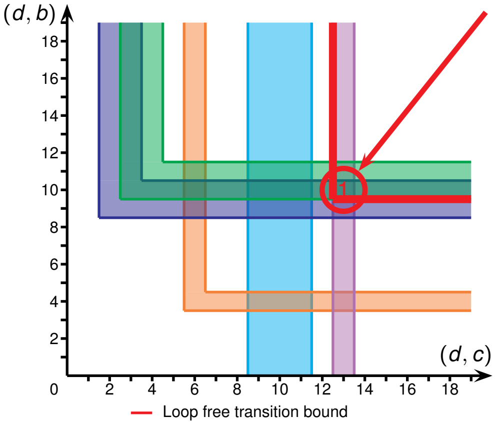
Defining Safe Weight Increment Sequences

Forward search:

1. c_3, c_4, c_5 *(greedy)*
2. c_1
3. c_2

Backward search:

1. c_2, c_4, c_5



A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

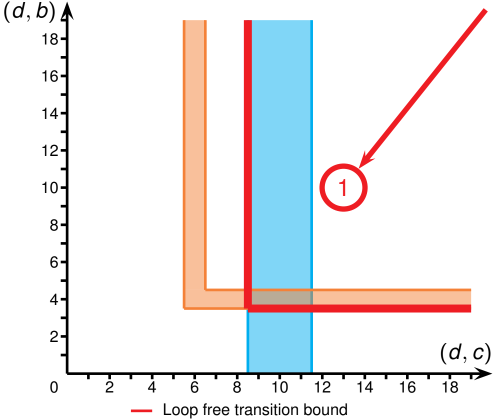
Defining Safe Weight Increment Sequences

Forward search:

1. c_3, c_4, c_5 *(greedy)*
2. c_1
3. c_2

Backward search:

1. c_2, c_4, c_5



A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

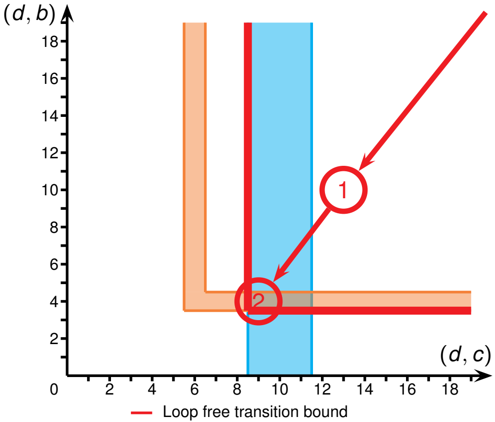
Defining Safe Weight Increment Sequences

Forward search:

1. c_3, c_4, c_5 *(greedy)*
2. c_1
3. c_2

Backward search:

1. c_2, c_4, c_5
2. c_1, c_3



A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Defining Safe Weight Increment Sequences

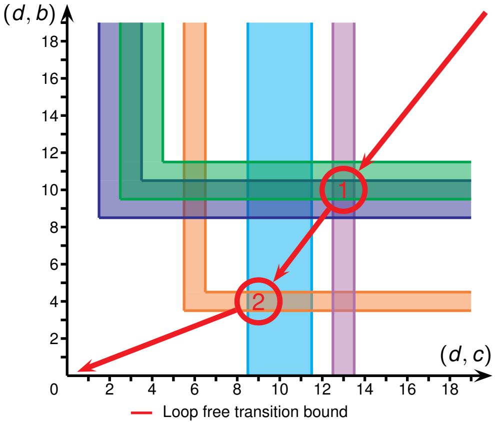
Forward search:

1. c_3, c_4, c_5 *(greedy)*
2. c_1
3. c_2

Backward search:

1. c_2, c_4, c_5
2. c_1, c_3

▷ Deterministic process



A weight sequence s avoids a loop L if and only if s contains at least one vector meeting the corresponding constraint.

Greedy backward algorithm (GBA)

Algorithm

1. For each loop L, add the corresponding constraint c to CS.
2. Add to the sequence S a greedy vector gv such that:

$$\forall i \in [1, |gv|], gv[i] = MAX\ (\underline{c}_1[i], \underline{c}_2[i], \ldots \underline{c}_n[i]) + 1$$

3. Remove from CS all constraints *met* by gv.

Repeat steps 2 *and* 3 *until there is no more constraints in CS.*

Greedy backward algorithm (GBA)

Algorithm

1. For each loop L, add the corresponding constraint c to CS.
2. Add to the sequence S a greedy vector gv such that:

$$\forall i \in [1, |gv|], gv[i] = MAX\ (\underline{c}_1[i], \underline{c}_2[i], \ldots \underline{c}_n[i]) + 1$$

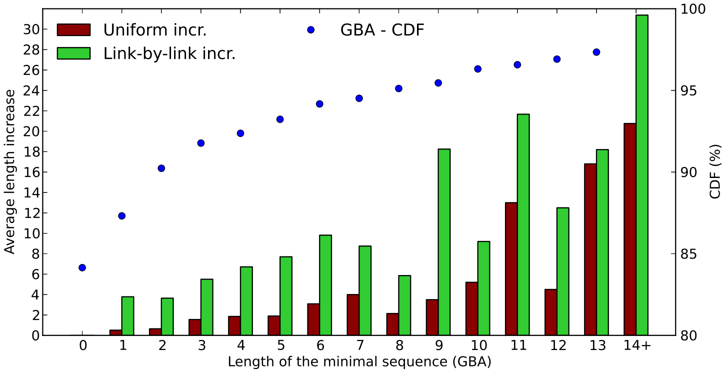
3. Remove from CS all constraints *met* by gv.

Repeat steps 2 *and* 3 *until there is no more constraints in CS.*

Theorem

Given a set of loop-constraints, GBA computes a minimal sequence of intermediate increments preventing convergence loops.

Sequence Lengths on a Large ISP Network



- 90% of the nodes requiring up to 3 intermediate steps
- Link-by-link sequences more than 200% longer

Conclusion

- Link shut problem
  - Minimal solution
  - Low time complexity
- Node shut problem
  - Minimal solution
  - Reasonable time complexity
  - Avoid flapping

Conclusion

- Link shut problem
  - ✓ Minimal solution
  - ▷ Low time complexity
- Node shut problem
  - ▷ Minimal solution
  - ▷ Reasonable time complexity
  - ▷ Avoid flapping

Conclusion

- Link shut problem
  - ✓ Minimal solution
  - ✓ Low time complexity (polynomial)
- Node shut problem
  - ▷ Minimal solution
  - ▷ Reasonable time complexity
  - ▷ Avoid flapping

Conclusion

- ✓ Link shut problem
  - ✓ Minimal solution
  - ✓ Low time complexity (polynomial)
- Node shut problem
  - ▷ Minimal solution
  - ▷ Reasonable time complexity
  - ▷ Avoid flapping

Conclusion

- ✓ Link shut problem
  - ✓ Minimal solution
  - ✓ Low time complexity (polynomial)

- Node shut problem
  - ✓ Minimal solution
  - ▷ Reasonable time complexity
  - ▷ Avoid flapping

Conclusion

- ✓ Link shut problem
  - ✓ Minimal solution
  - ✓ Low time complexity (polynomial)

- Node shut problem
  - ✓ Minimal solution
  - ✓ Reasonable time complexity (polynomial)
  - ▷ Avoid flapping

Conclusion

- ✓ Link shut problem
  - ✓ Minimal solution
  - ✓ Low time complexity (polynomial)

- Node shut problem
  - ✓ Minimal solution
  - ✓ Reasonable time complexity (polynomial)
  - ? Avoid flapping
    - ▷ Improve handling of flapping loops

What next?

- Theoretical extensions
  - Interactions with BGP and other routing protocols
  - Extension to multicast communications
  - Weight modifications on multiple independent links

- Experimental evaluations
  - Implementation and emulation with Quagga
  - Measurements on real networks (RENATER)

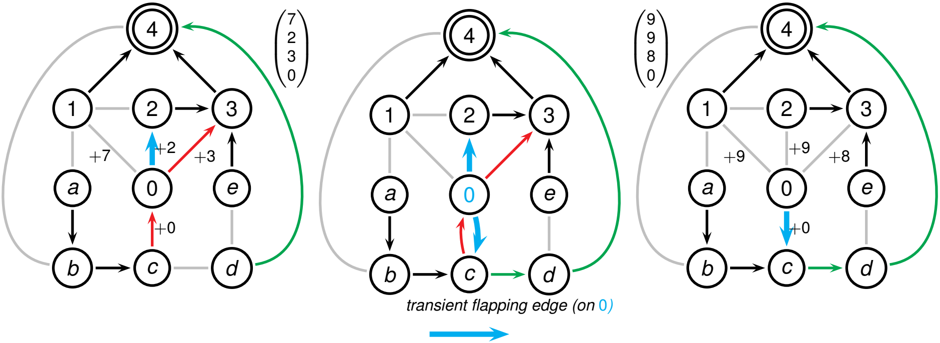
Thank you for your attention.

6 Appendix

Transient loop induced by route flapping

$\rightarrow$ $RSPDAG_1(4)$
Intermediate routing state towards 4
considering the first vector

$\rightarrow$ $RSPDAG_2(4)$
Intermediate routing state towards 4
considering the second vector



$$S_{GBA} = \begin{pmatrix}7\\2\\3\\0\end{pmatrix}, \begin{pmatrix}9\\9\\8\\0\end{pmatrix}$$

$$S_{FF1} = \begin{pmatrix}3\\2\\3\\0\end{pmatrix}, \begin{pmatrix}7\\4\\5\\0\end{pmatrix}, \begin{pmatrix}9\\9\\8\\0\end{pmatrix} \quad S_{FF2} = \begin{pmatrix}7\\2\\3\\0\end{pmatrix}, \begin{pmatrix}9\\9\\8\\3\end{pmatrix}$$

Global result table

Topology	#nodes / #edges	$S = \emptyset$	Uniform		Std GBA	
			$\lvert S\rvert \leq 5$	max	$\lvert S\rvert \leq 5$	max
Abilene	11 / 28	36.4 %	100 %	3	100 %	3
GEANT	22 / 72	63.6 %	100 %	5	100 %	3
ISP1	25 / 55	69.2 %	100 %	4	100 %	4
ISP2	55 / 195	81.5 %	94.4 %	7	100 %	3
ISP3	110 / 340	59.1 %	81.8 %	21	90.9 %	10
ISP4	140 / 410	67.4 %	85.8 %	21	92.9 %	10
ISP5	210 / 785	56.7 %	74.8 %	63	82.9 %	33
ISP6	1170 / 7240	84.2 %	92.1 %	147	93.2 %	57

GBA theory

Theorem

A weight sequence s avoids a loop L if and only if all pairs of successive vectors of s form a safe transition with respect to the constraint corresponding to L.

Theorem

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.

Lemma

At each iteration, GBA computes a vector v that meets at least one constraint not met before.

Problem

Constraint Minimal Meeting Problem (CMP): Given a set $cs = \{(\underline{c}_1, \bar{c}_1), \ldots, (\underline{c}_n, \bar{c}_n)\}$ *of loop-constraints, compute a minimal weight increment sequence which contains no unsafe transition for any constraint in cs.*

Theorem

Given a CMP instance I, GBA computes sequences that prevent convergence loops.

Lemma

Consider a CMP instance I. Let $s = (v_1 \ldots v_n)$ *be any sequence solving I, and let* $g = (g_1 \ldots g_m)$ *be the sequence computed by GBA on I, with possibly* $n \neq m$*. Then, the last respective vectors verify* $v_n \geq g_m$*.*

Lemma

Consider a CMP instance I. Let $s = (v_1 \ldots v_n)$ *be any sequence solving I, and let* $g = (g_1 \ldots g_m)$ *be the sequence computed by GBA on I, with possibly* $n \neq m$*. Then, all the constraints met by* v_n *(and possibly more) are also met by* g_m*.*

Theorem

The GBA algorithm finds a minimal sequence for any CMP instance I.

Theorem

GBA terminates in a number of main loop iterations which is polynomial with respect to the number of routers in the network.