From link shut to node shut:

Graceful operations in link-state routing networks

François CLAD, Pascal MERINDOL and Jean-Jacques PANSIOT

Team presentation
December 17th, 2012

1 Introduction

2 Transient loops

3 Link shut

4 Node shut

5 Conclusion

Some context

- Routing in providers' networks
  - Intra-domain routing
  - Link-state protocols : OSPF, IS-IS
- Frequent topological changes
  - Link or node addition, withdrawal or modification
  - . . . and as many convergence periods

1 Introduction

2 Transient loops

3 Link shut

4 Node shut

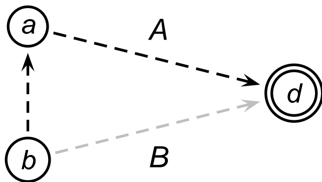
5 Conclusion

How do transient loops appear ?

Routers' update order is **not controlled.**
It depends on *LSA flooding* and *RIB/FIB update* times.

- Initially, both a and b reach d through a.

Routes towards d :

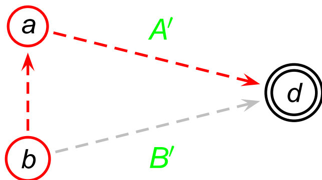


How do transient loops appear ?

Routers' update order is **not controlled.**
It depends on *LSA flooding* and *RIB/FIB update* times.

- 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* :



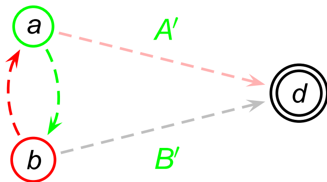
Old : $A << B$
New : $A' >> B'$

How do transient loops appear ?

Routers' update order is **not controlled.**
It depends on *LSA flooding* and *RIB/FIB update* times.

- 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* :



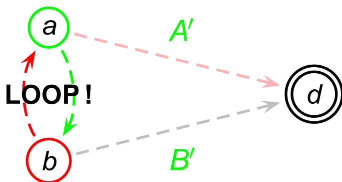
Old : $A << B$
New : $A' >> B'$

How do transient loops appear ?

Routers' update order is **not controlled.**
It depends on *LSA flooding* and *RIB/FIB update* times.

- 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 :

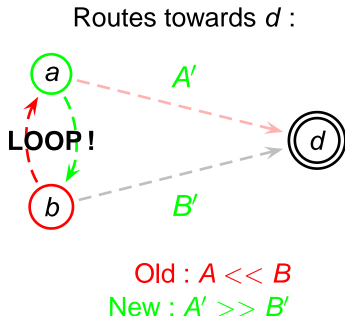


Old : $A << B$
New : $A' >> B'$

How do transient loops appear ?

Routers' update order is **not controlled.**
It depends on *LSA flooding* and *RIB/FIB update* times.

- 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

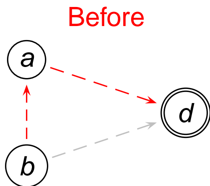


How to detect them ?

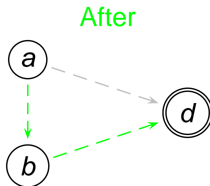
For a given destination (e.g. *d*) :

1. Compute routes before and after the change.

Before



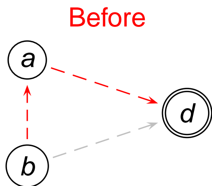
After

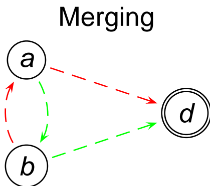


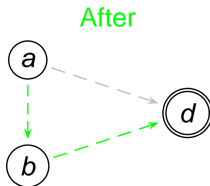
How to detect them ?

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

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



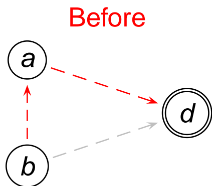


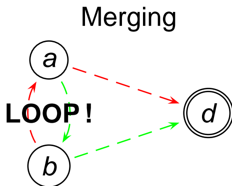


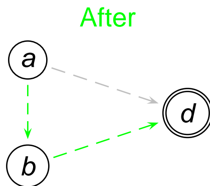
How to detect them ?

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.



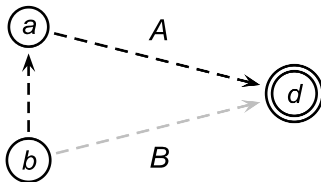




How to prevent them ?

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

- Initially, both a and b reach d through a.



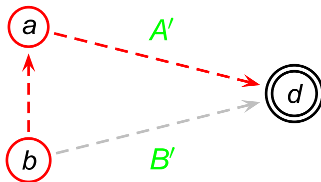
$$A + w(b, a) < B$$

P. Francois and O. Bonaventure, “Avoiding Transient Loops During the Convergence of Link-State Routing Protocols”, *IEEE/ACM Transactions on Networking*, volume 15, pages 1280-1292, December 2007.

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...



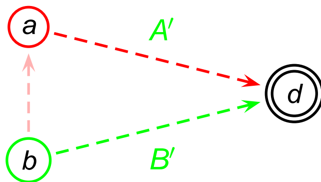
Old : $A + w(b, a) < B$
New : $A' > w(a, b) + B'$

P. Francois and O. Bonaventure, “Avoiding Transient Loops During the Convergence of Link-State Routing Protocols”, *IEEE/ACM Transactions on Networking*, volume 15, pages 1280-1292, December 2007.

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 : $A + w(b, a) < B$

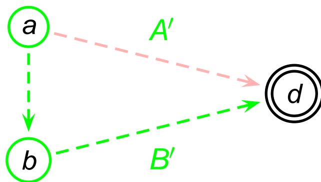
New : $A' > w(a, b) + B'$

P. Francois and O. Bonaventure, “Avoiding Transient Loops During the Convergence of Link-State Routing Protocols”, *IEEE/ACM Transactions on Networking*, volume 15, pages 1280-1292, December 2007.

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 : $A + w(b, a) < B$

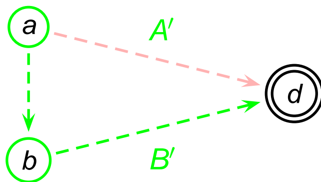
New : $A' > w(a, b) + B'$

P. Francois and O. Bonaventure, “Avoiding Transient Loops During the Convergence of Link-State Routing Protocols”, *IEEE/ACM Transactions on Networking*, volume 15, pages 1280-1292, December 2007.

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 : $A + w(b, a) < B$
New : $A' > w(a, b) + B'$

P. Francois and O. Bonaventure, “Avoiding Transient Loops During the Convergence of Link-State Routing Protocols”, *IEEE/ACM Transactions on Networking*, volume 15, pages 1280-1292, December 2007.

Progressive update

Basic idea

Split up the change into a sequence of smaller modifications, such that **each one is loop free.**

Objectives

Compute a sequence of intermediate updates, such that :

- No transient loop may appear during the transition between two consecutive updates.
- Each intermediate update prevents at least one cycle.

Challenge

Minimal operational impact (sequences of minimal size)

Illustration : path increment sequence

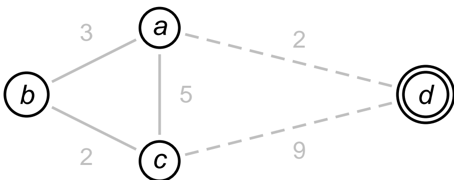


Illustration : path increment sequence

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

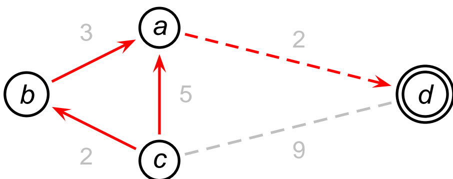


Illustration : path increment sequence

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

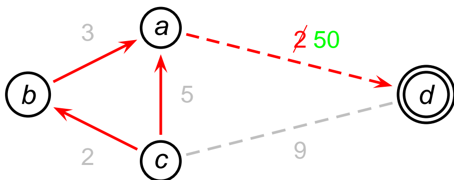


Illustration : path increment sequence

- 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 . . .

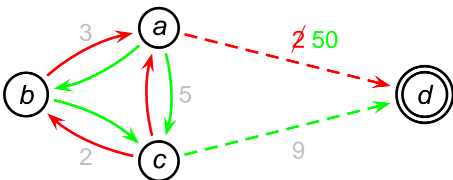


Illustration : path increment sequence

- 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.

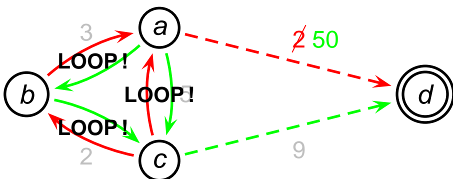


Illustration : path increment sequence

- 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 progressive increments :

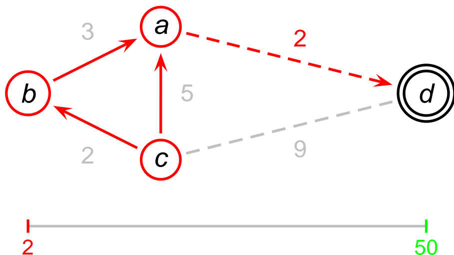


Illustration : path increment sequence

- 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 progressive increments :

- Node c could update first

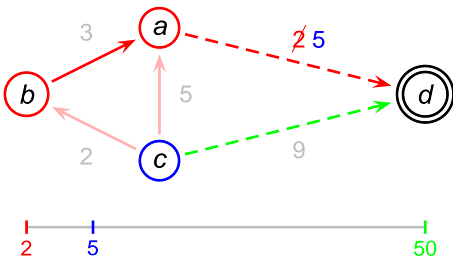


Illustration : path increment sequence

- 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 progressive increments :

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

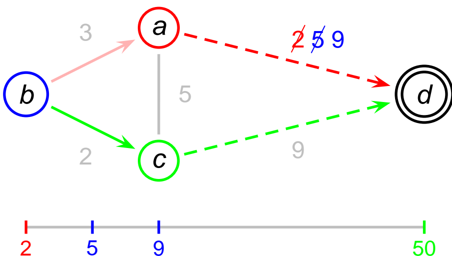


Illustration : path increment sequence

- 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 progressive increments :

- Node *c* could update first
- Then *b* and *a*.

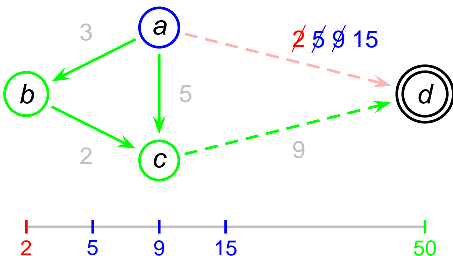


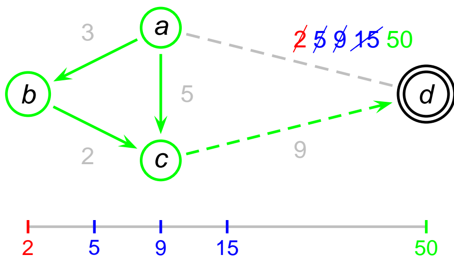
Illustration : path increment sequence

- 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 progressive increments :

- Node *c* could update first
- Then *b* and *a*.

So that the transition to 50 will be loop free.



1 Introduction

2 Transient loops

3 Link shut

4 Node shut

5 Conclusion

Case of a link shut (withdrawal)[1]

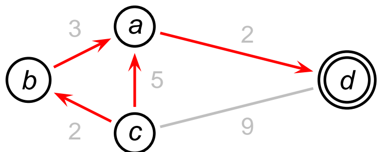
1. Compute a list of **affected destinations**
2. Compute *new* paths toward these nodes (after removal)
3. Extract **destination oriented** metric sequences
4. Merge and reduce them to build a **global** sequence

---

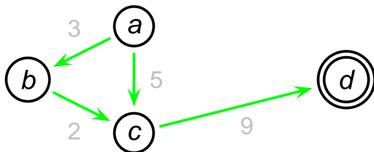
1. The same algorithms may be used for any other kind of modification on a single link (addition, weight increment or decrement).

Destination oriented sequences : Δ values

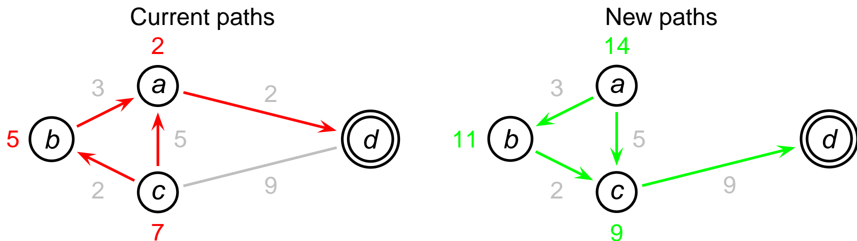
Current paths



New paths

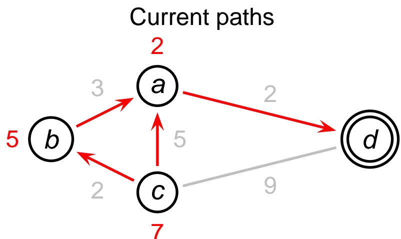


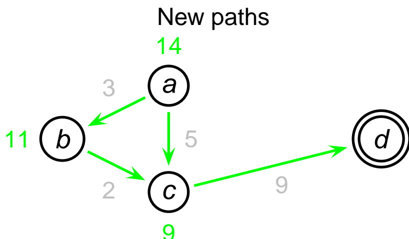
Destination oriented sequences : Δ values



- Retrieve distances from each node to the destination

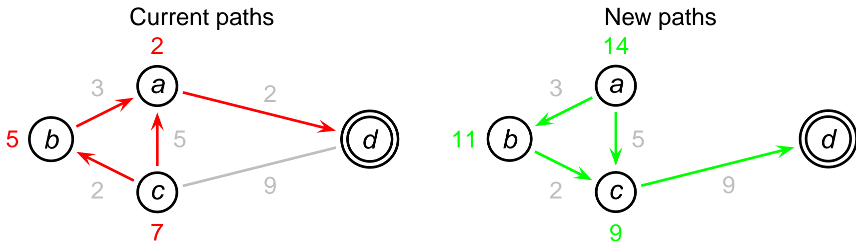
Destination oriented sequences : Δ values





- Retrieve distances from each 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

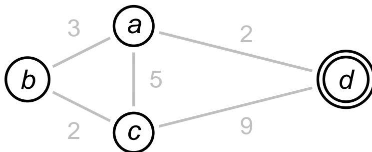


- Retrieve distances from each node to the destination
- Compute the difference (Δ) between new and old distances
  - $\Delta(a) = 14 - 2 = 12$
  - $\Delta(b) = 11 - 5 = 6$
  - $\Delta(c) = 9 - 7 = 2$

▷ 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

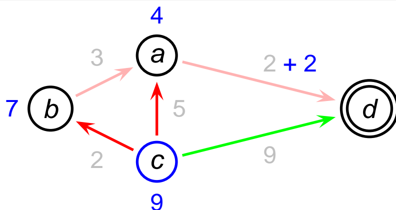
While in **ECMP state**, a node uses both its old and new routes towards the destination.



- Δ sequence : $\{2, 6, 12\}$

Destination oriented sequences : ECMP state

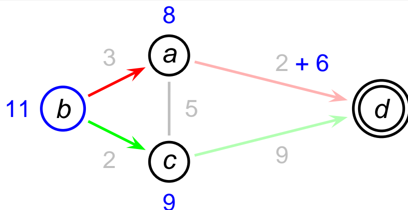
While in **ECMP state**, a node uses both its old and new routes towards the destination.



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

Destination oriented sequences : ECMP state

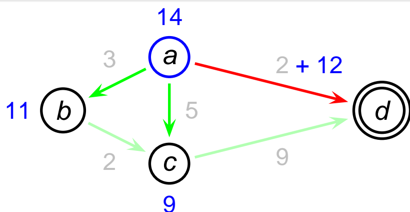
While in **ECMP state**, a node uses both its old and new routes towards the destination.



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

Destination oriented sequences : ECMP state

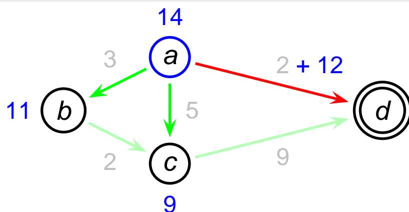
While in **ECMP state**, a node uses both its old and new routes towards the destination.



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

Destination oriented sequences : ECMP state

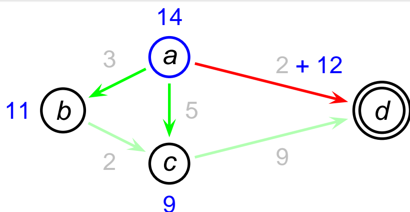
While in **ECMP state**, a node uses both its old and new routes towards the destination.



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

Destination oriented sequences : ECMP state

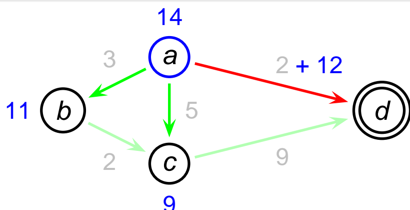
While in **ECMP state**, a node uses both its old and new routes towards the destination.



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

Destination oriented sequences : ECMP state

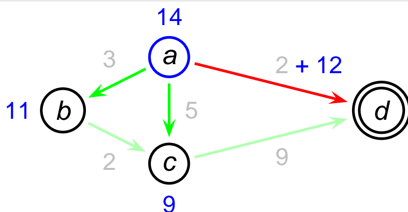
While in **ECMP state**, a node uses both its old and new routes towards the destination.



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

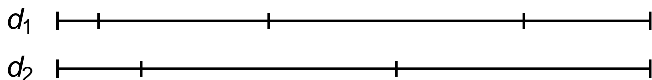
Destination oriented sequences : ECMP state

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

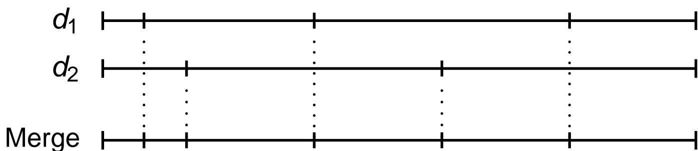


- Δ sequence : $\{2, 6, 12\}$ relative to $w(a, d)$
  - ▷ First values such that the nodes use their new path(s)
  - ▷ **Does not prevent transient loops**
- Increment sequence $(\Delta + 1)$: $\{3, 7, 13\}$ relative to $w(a, d)$
  - ▷ First values such that the nodes use **only** their new path(s)
-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

Global metric sequences

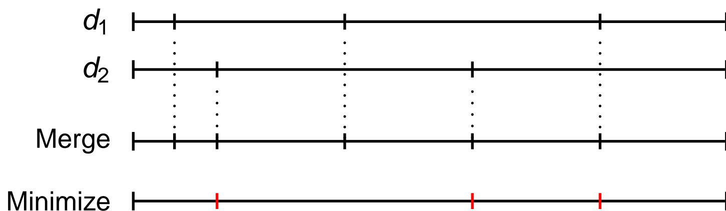


Global metric sequences



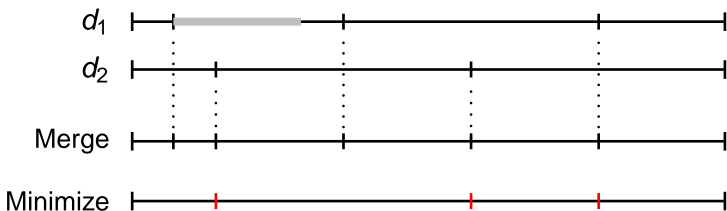
- Merge all destination oriented sequences
  - ▷ Prevent transient loops for all destinations
  - ▷ May contain unnecessary metrics

Global metric sequences



- Merge all destination oriented sequences
  - Prevent transient loops for all destinations
  - May contain unnecessary metrics
- Sequentially walk through the global sequence and prune as many intermediate metrics as possible.
  - Greedy algorithm looking for possible loops at each step
  - Ensure the minimality in terms of sequence size

Global metric sequences



- Merge all destination oriented sequences
  - Prevent transient loops for all destinations
  - May contain unnecessary metrics
- Sequentially walk through the global sequence and prune as many intermediate metrics as possible.
  - Greedy algorithm looking for possible loops at each step
  - Ensure the minimality in terms of sequence size

1 Introduction

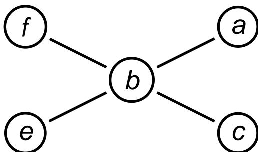
2 Transient loops

3 Link shut

4 Node shut

5 Conclusion

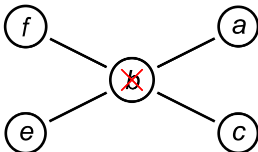
Case of a node shut[2]



- Shut each outgoing link of the node
- Increasing the weight of one link may shift the traffic to an alternate link
  - ▷ Flapping issues
- Several link updates in a single signalisation packet (LSA)
  - ▷ Vectorial increments

2. As for a link, the same procedure may be used for adding a node.

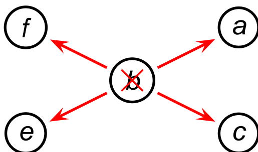
Case of a node shut[2]



- Shut each outgoing link of the node
- Increasing the weight of one link may shift the traffic to an alternate link
  - ▷ Flapping issues
- Several link updates in a single signalisation packet (LSA)
  - ▷ Vectorial increments

2. As for a link, the same procedure may be used for adding a node.

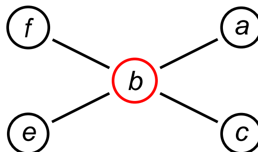
Case of a node shut[2]



- Shut each outgoing link of the node
- Increasing the weight of one link may shift the traffic to an alternate link
  - ▷ Flapping issues
- Several link updates in a single signalisation packet (LSA)
  - ▷ Vectorial increments

2. As for a link, the same procedure may be used for adding a node.

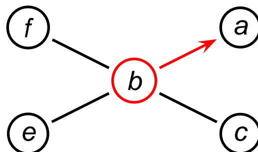
A first approach : link-by-link



- Proof of existence
- Shut outgoing links one-by-one using the *link shut* approach
- Different orders may produce sequences of different sizes

+ One algorithm for link and node shut
– Multiple traffic shifts (flapping), far from being minimal

A first approach : link-by-link

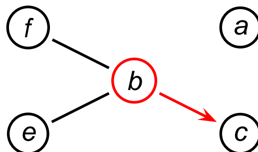


- Proof of existence
- Shut outgoing links one-by-one using the *link shut* approach
- Different orders may produce sequences of different sizes

\+ One algorithm for link and node shut

– Multiple traffic shifts (flapping), far from being minimal

A first approach : link-by-link

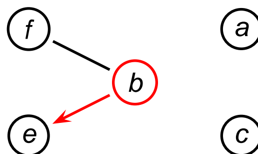


- Proof of existence
- Shut outgoing links one-by-one using the *link shut* approach
- Different orders may produce sequences of different sizes

+ One algorithm for link and node shut

– Multiple traffic shifts (flapping), far from being minimal

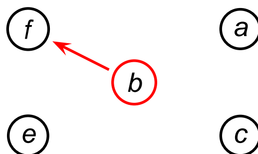
A first approach : link-by-link



- Proof of existence
- Shut outgoing links one-by-one using the *link shut* approach
- Different orders may produce sequences of different sizes

+ One algorithm for link and node shut
– Multiple traffic shifts (flapping), far from being minimal

A first approach : link-by-link



- Proof of existence
- Shut outgoing links one-by-one using the *link shut* approach
- Different orders may produce sequences of different sizes

\+ One algorithm for link and node shut

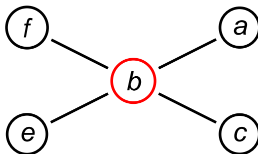
– Multiple traffic shifts (flapping), far from being minimal

A first approach : link-by-link

- Proof of existence
- Shut outgoing links one-by-one using the *link shut* approach
- Different orders may produce sequences of different sizes

+ One algorithm for link and node shut
– Multiple traffic shifts (flapping), far from being minimal

A smarter way : uniform increments

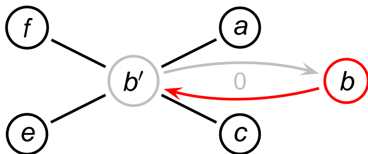


- **Uniform increments** on outgoing links
- Same algorithm as for the link shut

\+ Good computing performances, no traffic shift

– Still not minimal

A smarter way : uniform increments

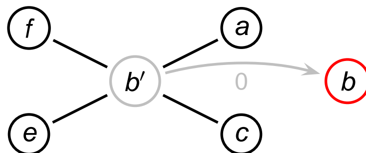


- **Uniform increments** on outgoing links
- Same algorithm as for the link shut

\+ Good computing performances, no traffic shift

– Still not minimal

A smarter way : uniform increments

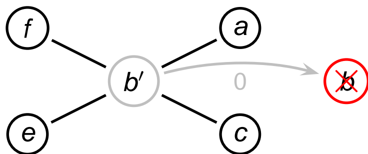


- **Uniform increments** on outgoing links
- Same algorithm as for the link shut

\+ Good computing performances, no traffic shift

– Still not minimal

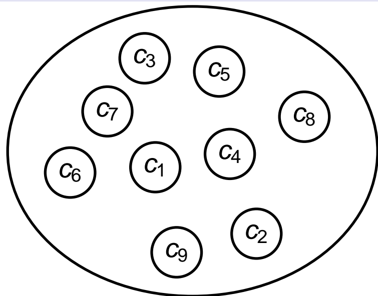
A smarter way : uniform increments



- **Uniform increments** on outgoing links
- Same algorithm as for the link shut

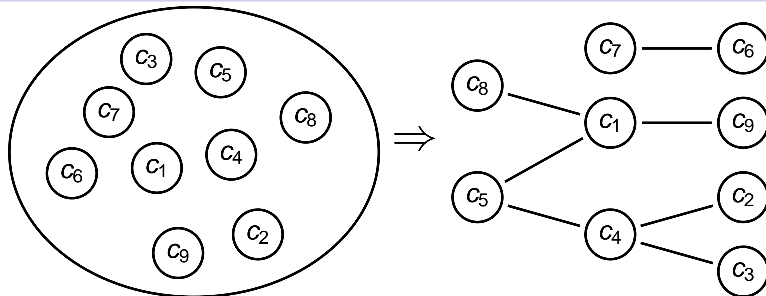
+ Good computing performances, no traffic shift
– Still not minimal

Towards minimality : Greedy Backward Algorithm



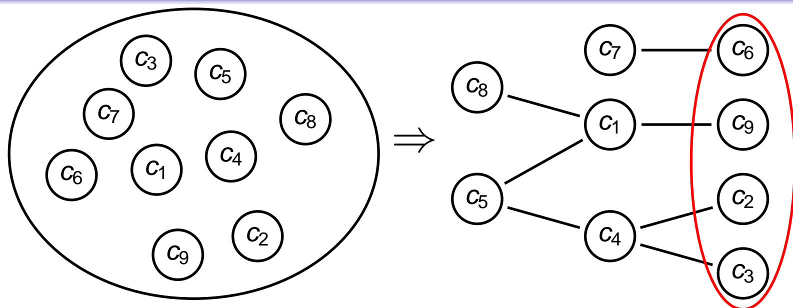
- Enumerate every transient cycle

Towards minimality : Greedy Backward Algorithm



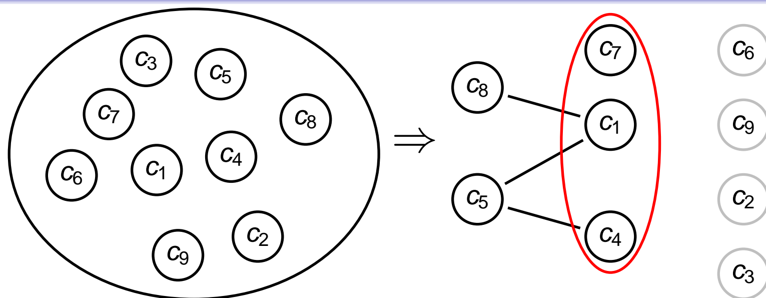
- Enumerate every transient cycle
- Order these cycles in a dependency DAG (partial order)

Towards minimality : Greedy Backward Algorithm



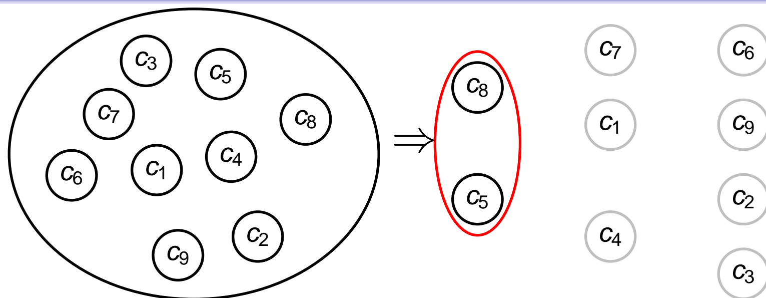
- Enumerate every transient cycle
- Order these cycles in a dependency DAG (partial order)
- Build intermediate vector increments

Towards minimality : Greedy Backward Algorithm



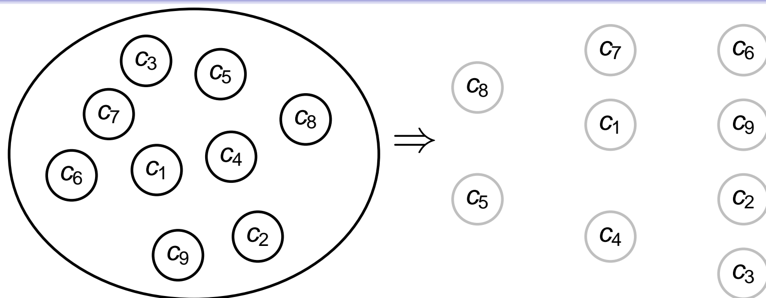
- Enumerate every transient cycle
- Order these cycles in a dependency DAG (partial order)
- Build intermediate vector increments

Towards minimality : Greedy Backward Algorithm



- Enumerate every transient cycle
- Order these cycles in a dependency DAG (partial order)
- Build intermediate vector increments

Towards minimality : Greedy Backward Algorithm

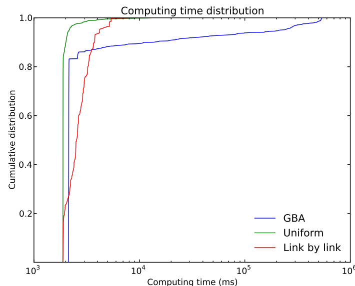


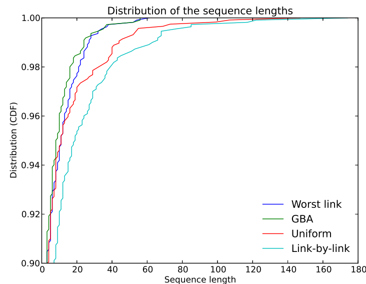
- Enumerate every transient cycle
- Order these cycles in a dependency DAG (partial order)
- Build intermediate vector increments

+ Minimal sequence size

− Path flapping, high computing times

Comparative evaluation





- Close results for most nodes
- Differences appear on *worst cases*

1 Introduction

2 Transient loops

3 Link shut

4 Node shut

5 Conclusion

Conclusion

- ✓ Link shut
  - ✓ Minimal solution
  - ✓ Low time complexity

- Node shut – work in progress
  - ✓ Minimal solution
  - ▷ Improve time performances
  - ▷ Avoid flapping

Thank you for your attention.