On the Resiliency of Randomized Routing Against Multiple Edge Failures

Slobodan Mitrović (EPFL)

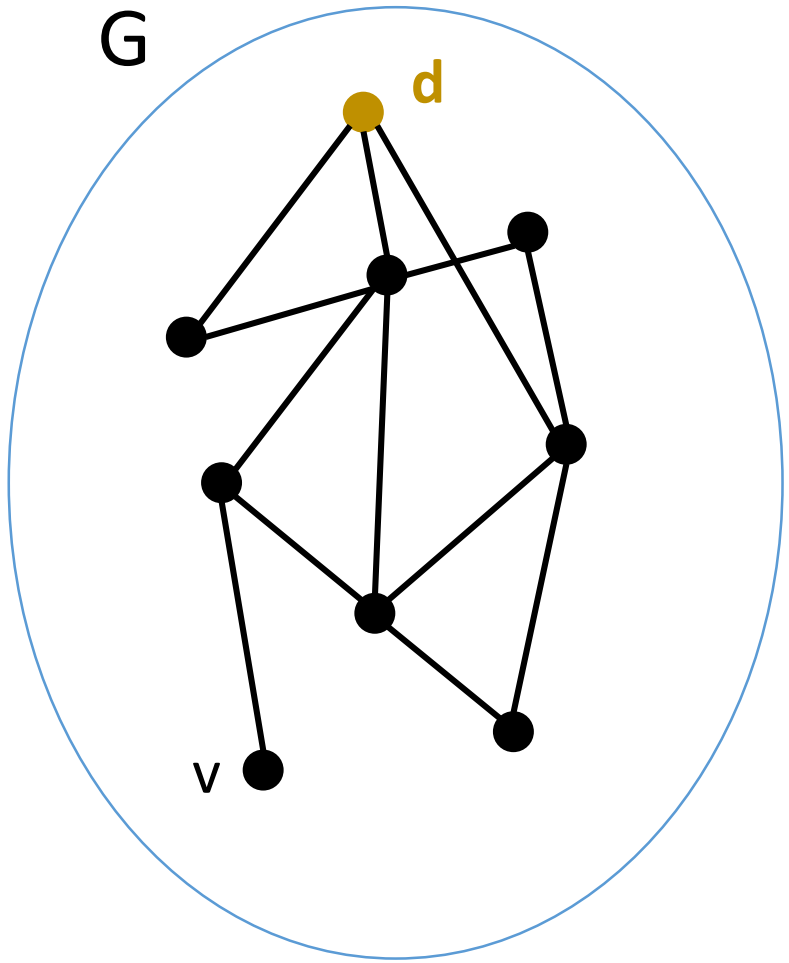
Joint work with M. Chiesa, A. Gurtov, A. Mądry, I. Nikolaevskiy, M. Schapira, and S. Shenker

Network routing

Given:

- a network *G = (V, E)*
- a target vertex **d** of *V*

Goal: deliver a packet from *v* to **d**



Network routing – Many different settings

1. per-destination
2. per-incoming-port
3. per-source-destination
4. packet-header rewriting
5. packet duplication
6. dynamic
7. …

Two important properties:

- being resilient, and
- fast computation (static and local routing scheme)

Network routing – The problem we study

Given:

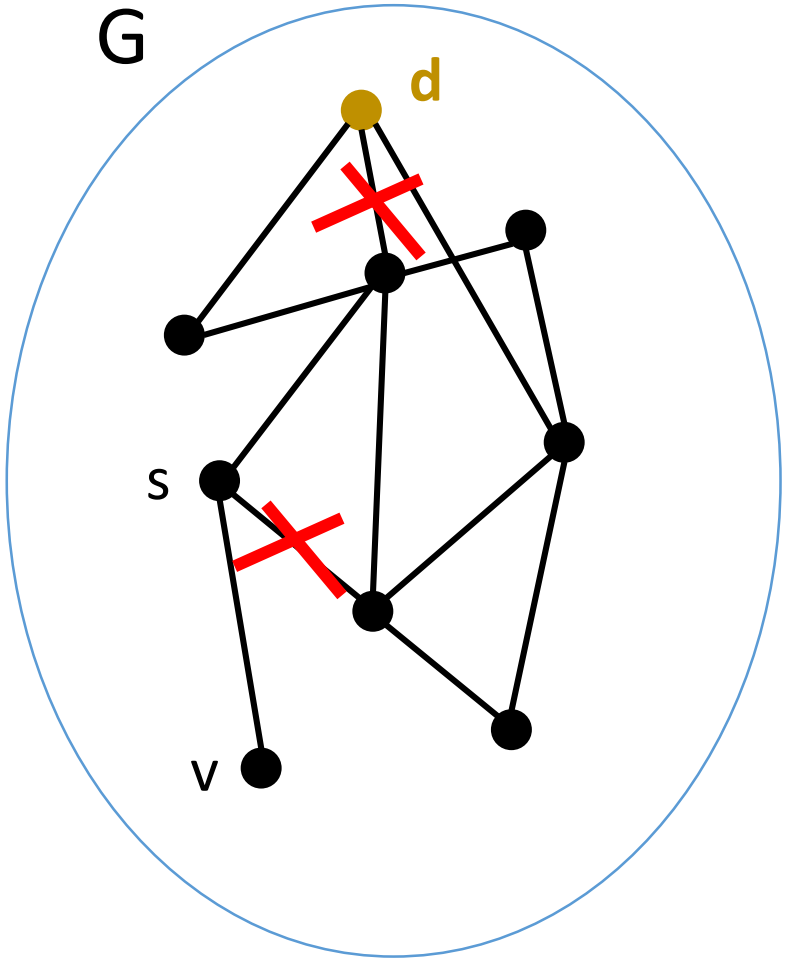
- a network *G = (V, E)*, a target vertex **d**
- a parameter *c*

Goal: Find a per-destination *static routing scheme* that delivers a packet from any source *s* to **d**

subject to: at most *c* links of *G* are failed

with property: routing is local (no packet-header rewriting; no broadcasting)

We say such scheme is *c-resilient*.

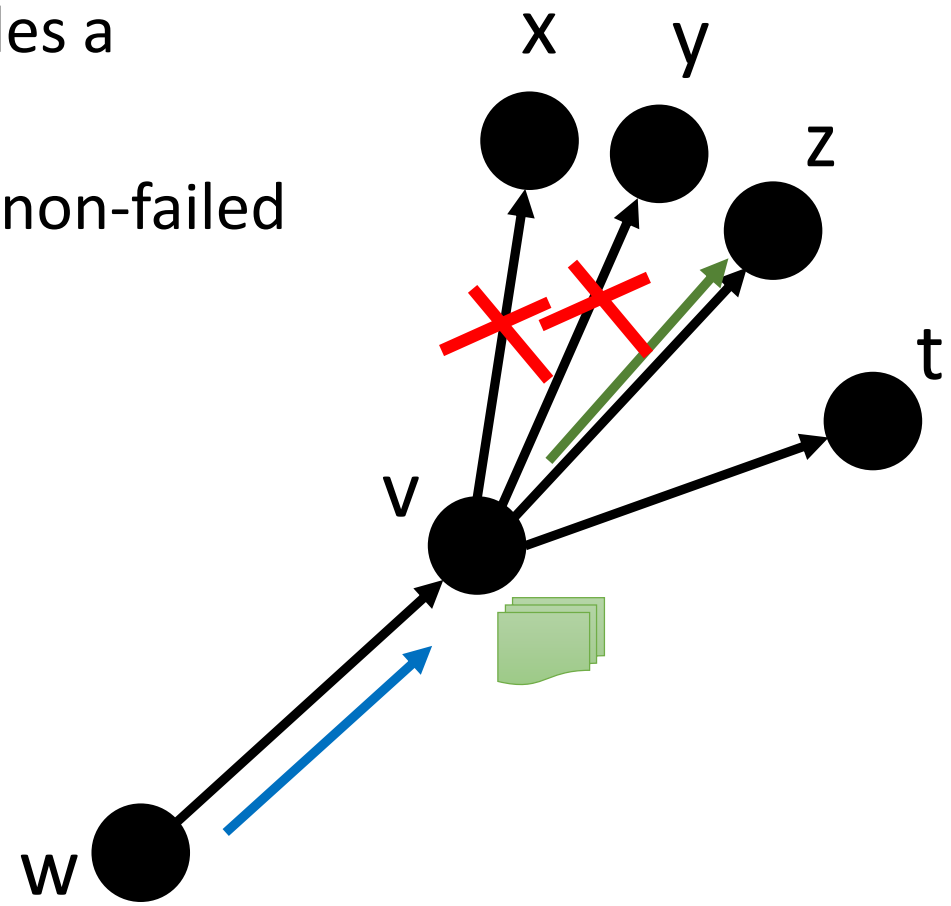


Local routing decisions

- Each vertex *v* has a list for each incoming link
- For incoming link from *w*, table provides a permutation of outgoing links
- The routing is continued through first non-failed link in the list

v:

x:	y	z	t	
y:	t	x	w	z
w:	x	y	z	t
z:	t	z	x	
t:	w			



Example

s:

start:	u

u:

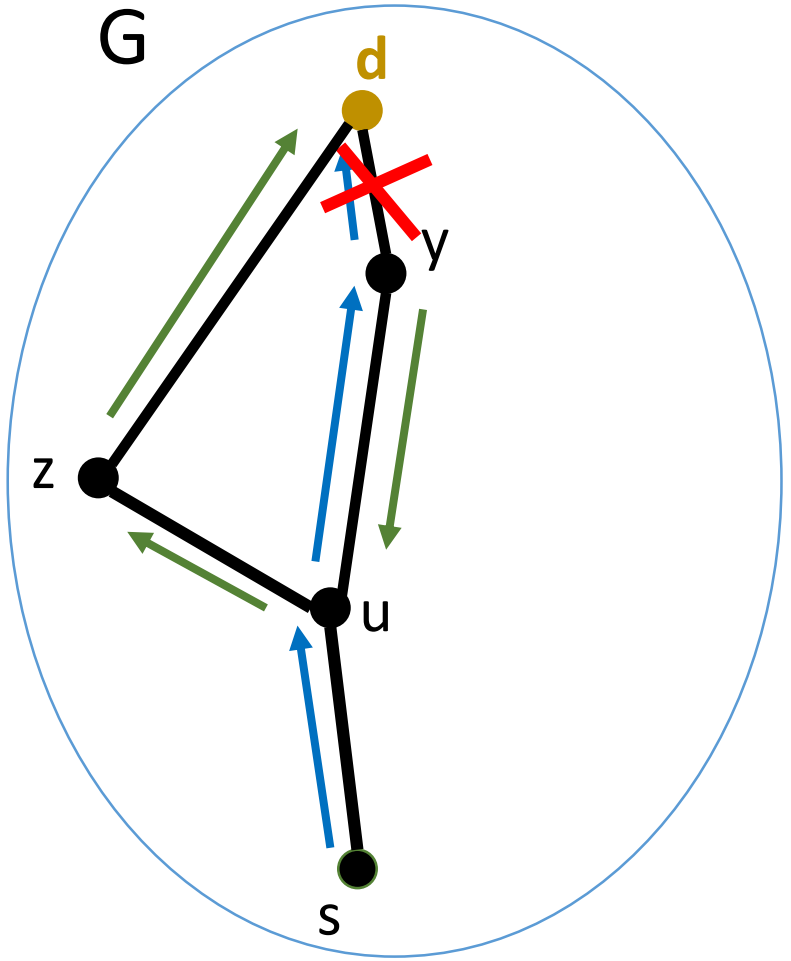
s:	y
y:	z

y:

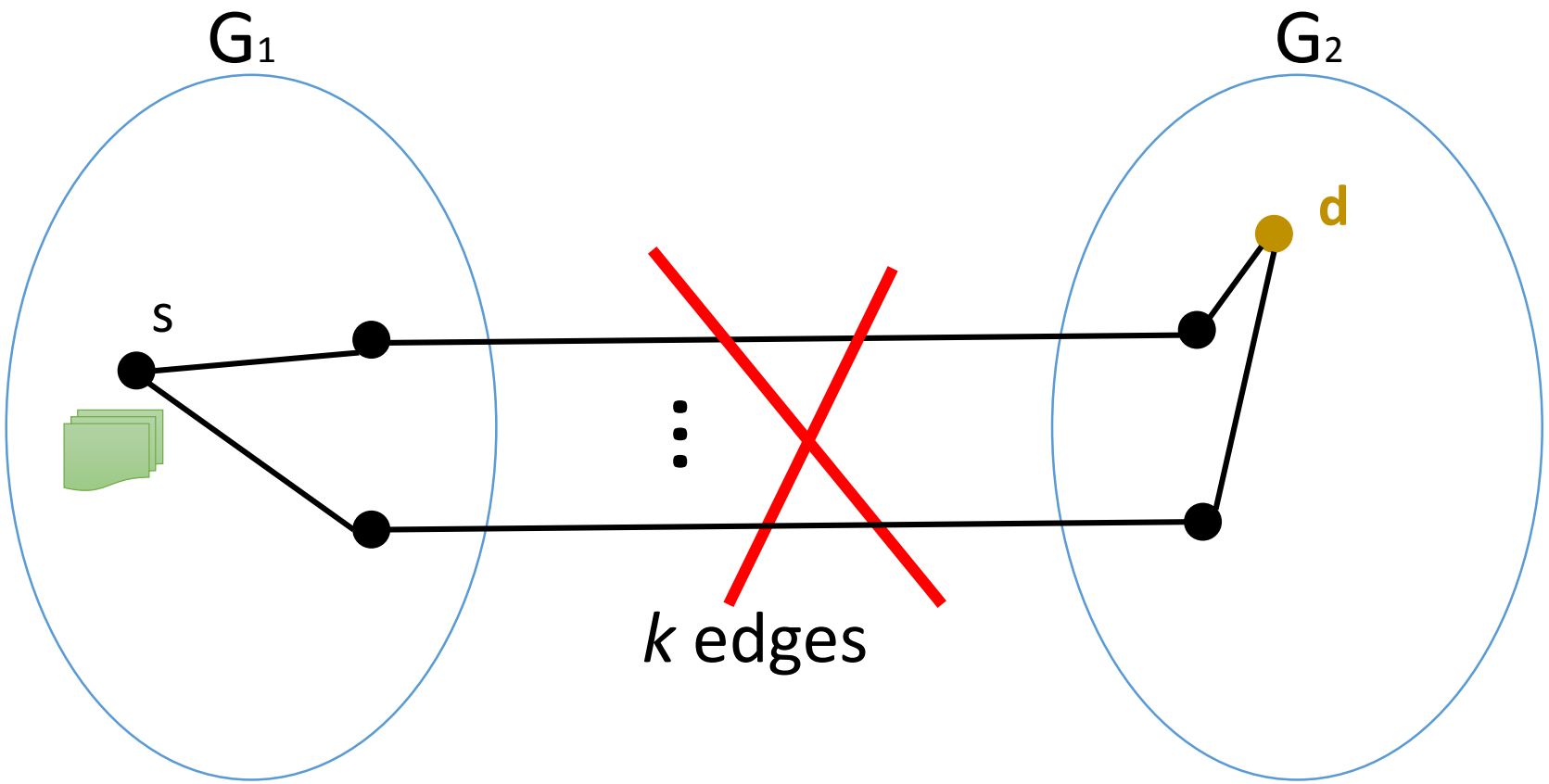
u:	d	u

z:

u:	d



We need *k*-connectivity



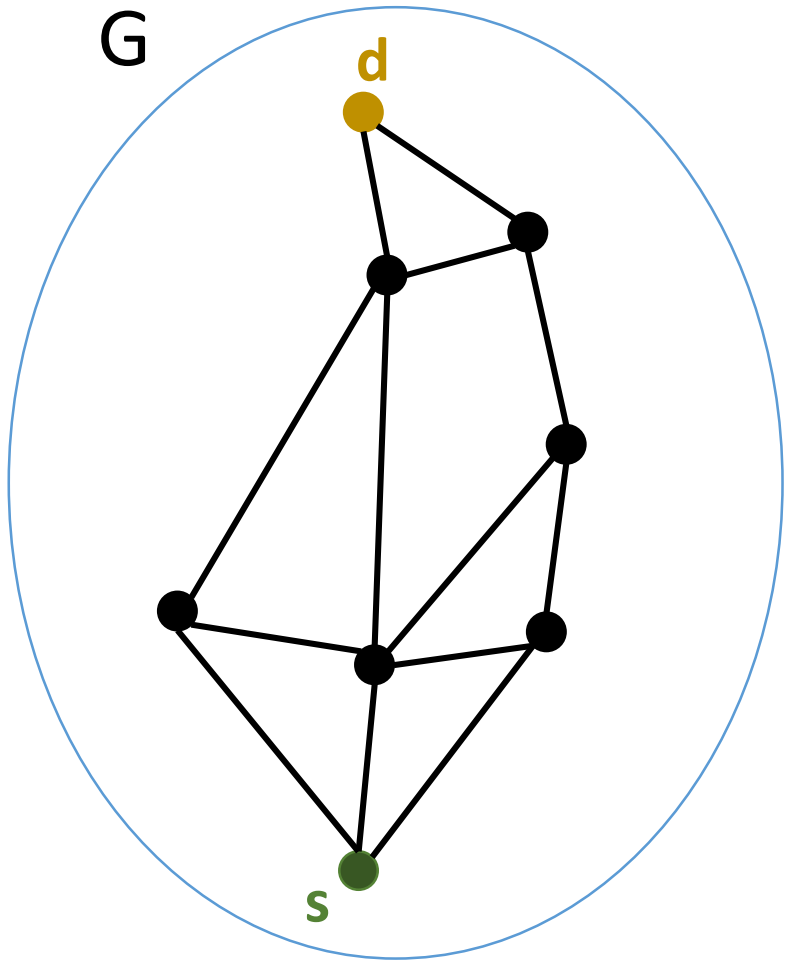
Necessary condition: *c*-resilience requires $k > c$.

Big challenge: Is $k > c$ sufficient too?

(from now, assume c < k)

Attempt 1 – Route along edge-disjoint paths

- [*Menger's theorem*] Between any u and v of G, there are at least k-edge disjoint paths.

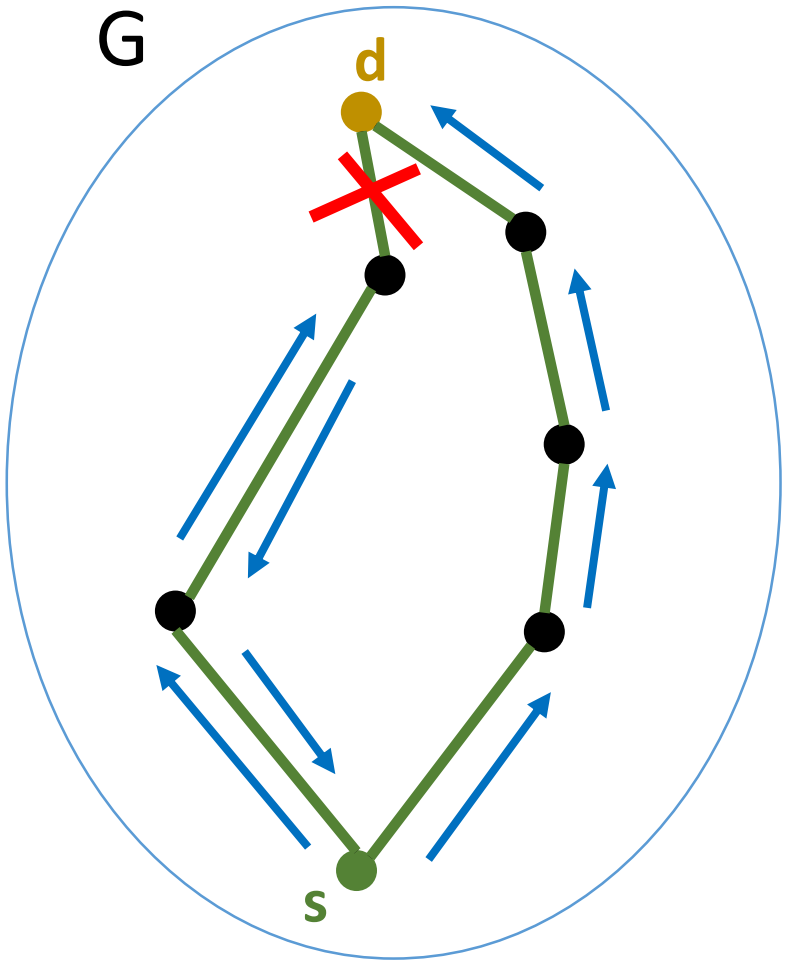


Attempt 1 – Route along edge-disjoint paths

- [*Menger's theorem*] Between any u and v of G, there are at least k-edge <u>disjoint</u> paths.

1. Find path packing
2. Route along paths
3. On failed edge, retreat back to s, and choose the next path

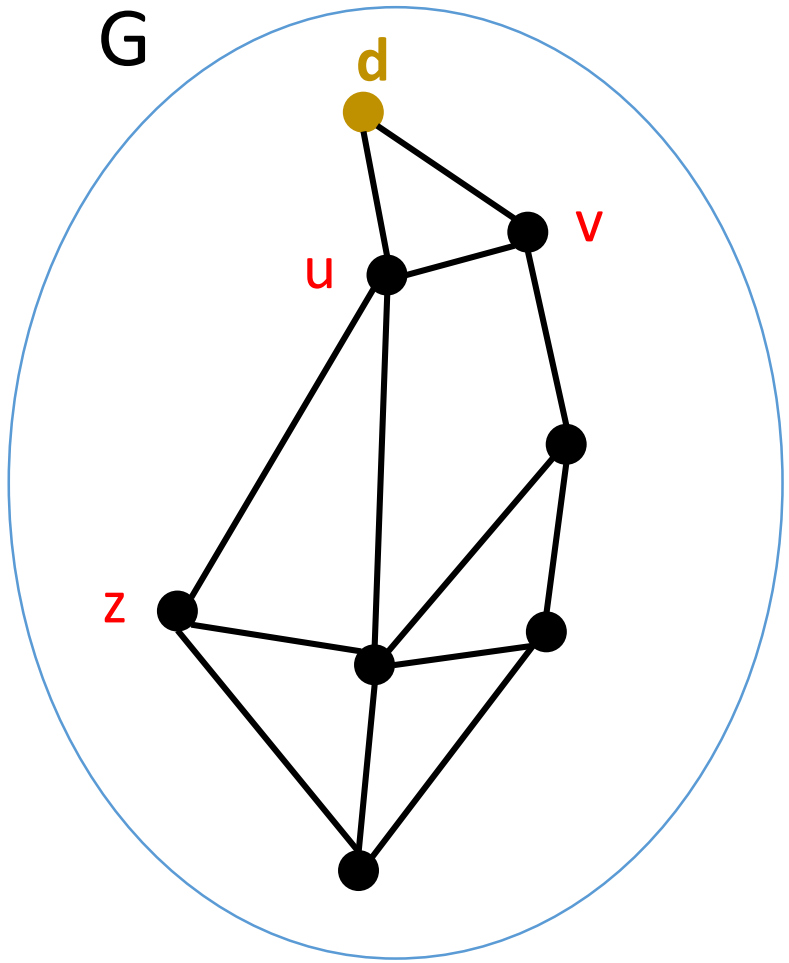
- But, <u>this is not satisfactory</u>, because the table is source-**d** based.



Attempt 2 – Packing arborescences

[*Edmonds, 1973*] A k-edge-connected graph G contains k arc-disjoint d-rooted arborescences.

1. Find arborescences packing
2. Route along arborescences
3. On failed edge, choose the next available arborescence (*circular routing*)



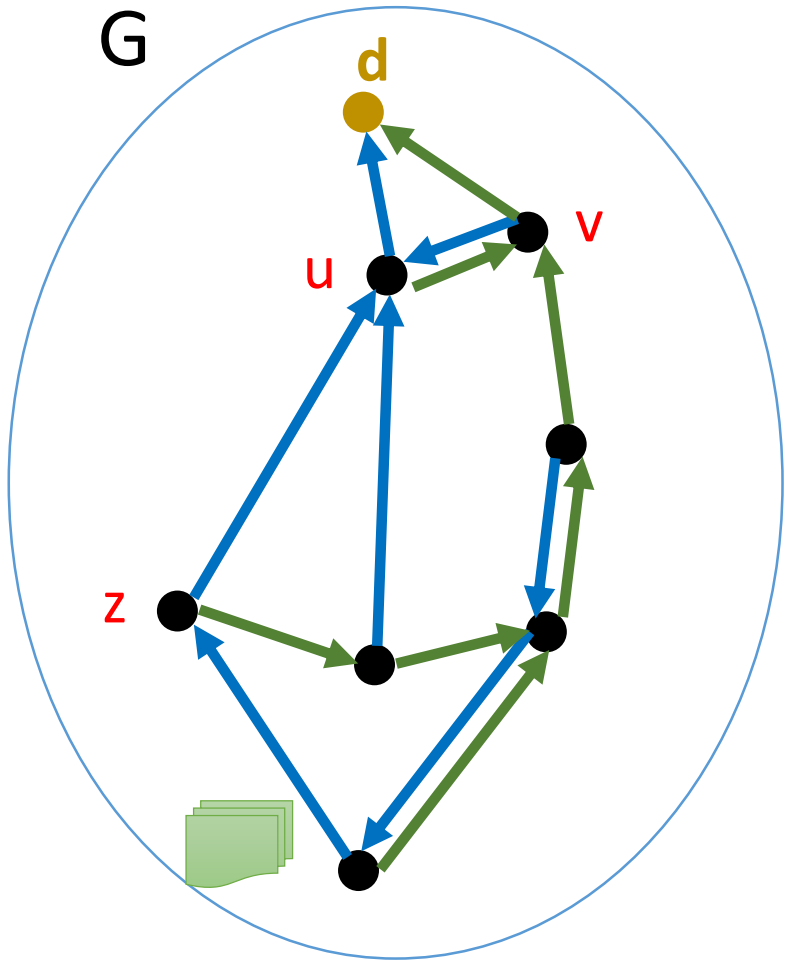
[EGR-INFOCOM14]
[CNMPGSS-INFOCOM16]

Attempt 2 – Packing arborescences

[*Edmonds, 1973*] A k-edge-connected graph G contains k arc-disjoint d-rooted arborescences.

1. Find arborescences packing
2. Route along arborescences
3. On failed edge, choose the next available arborescence (*circular routing*)

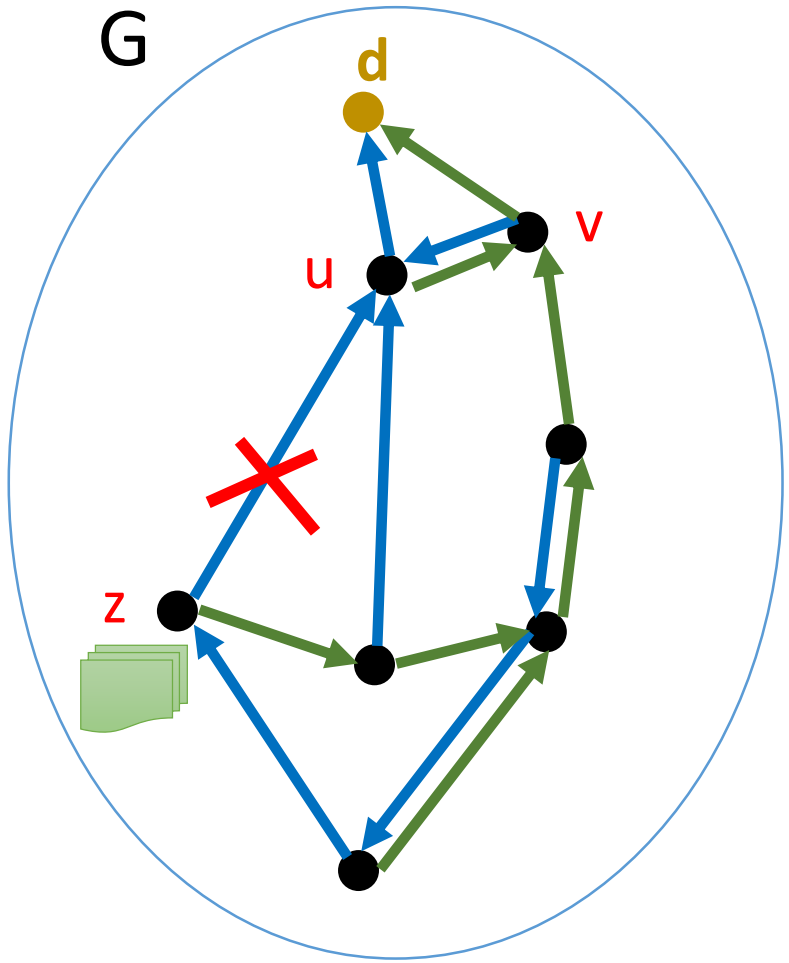
[EGR-INFOCOM14]
[CNMPGSS-INFOCOM16]



Attempt 2 – Packing arborescences

[*Edmonds, 1973*] A k-edge-connected graph G contains k arc-disjoint d-rooted arborescences.

1. Find arborescences packing
2. Route along arborescences
3. On failed edge, choose the next available arborescence (*circular routing*)



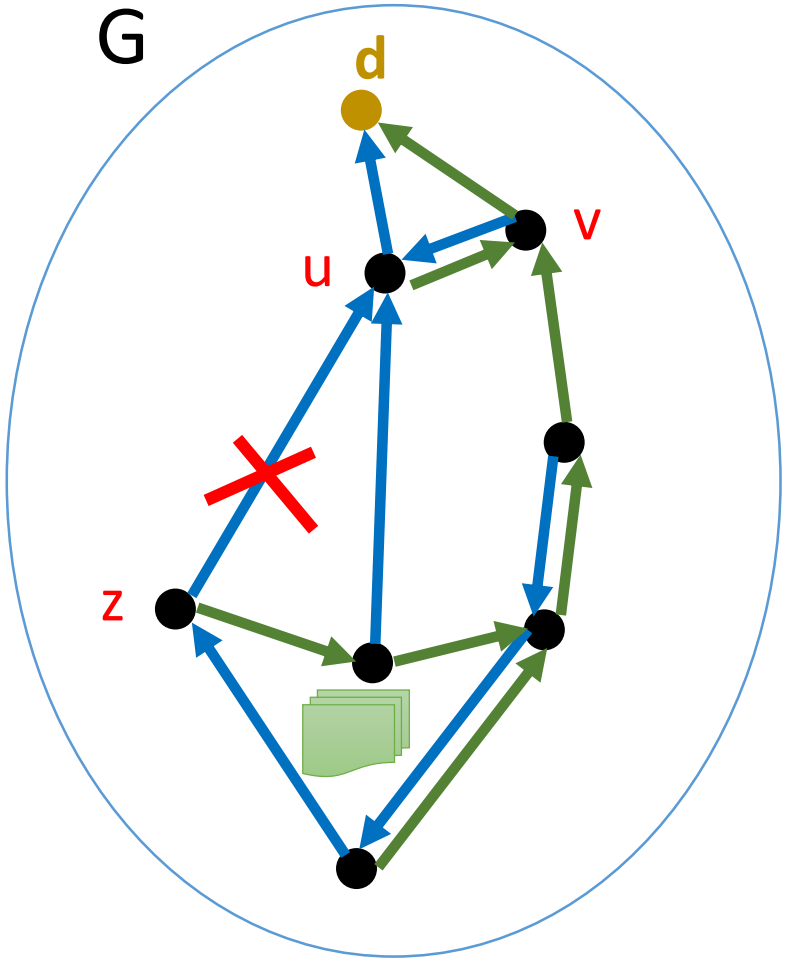
[EGR-INFOCOM14]
[CNMPGSS-INFOCOM16]

Attempt 2 – Packing arborescences

[*Edmonds, 1973*] A k-edge-connected graph G contains k arc-disjoint d-rooted arborescences.

1. Find arborescences packing
2. Route along arborescences
3. On failed edge, choose the next available arborescence (*circular routing*)

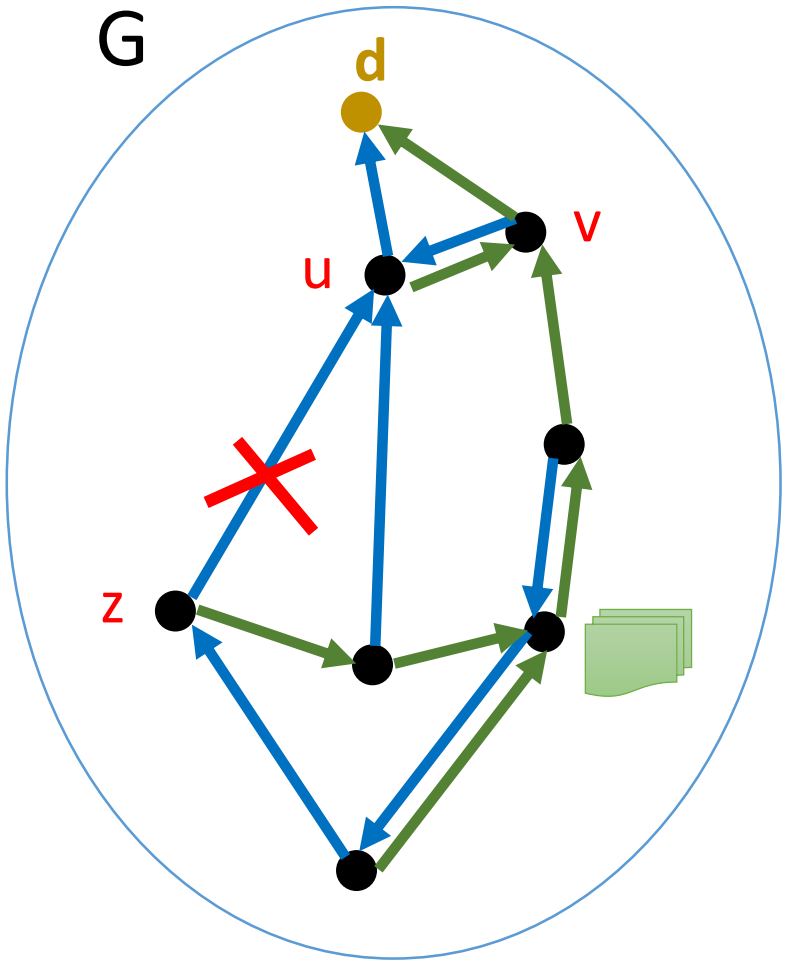
[EGR-INFOCOM14]
[CNMPGSS-INFOCOM16]



Attempt 2 – Packing arborescences

[*Edmonds, 1973*] A k-edge-connected graph G contains k arc-disjoint d-rooted arborescences.

1. Find arborescences packing
2. Route along arborescences
3. On failed edge, choose the next available arborescence (*circular routing*)

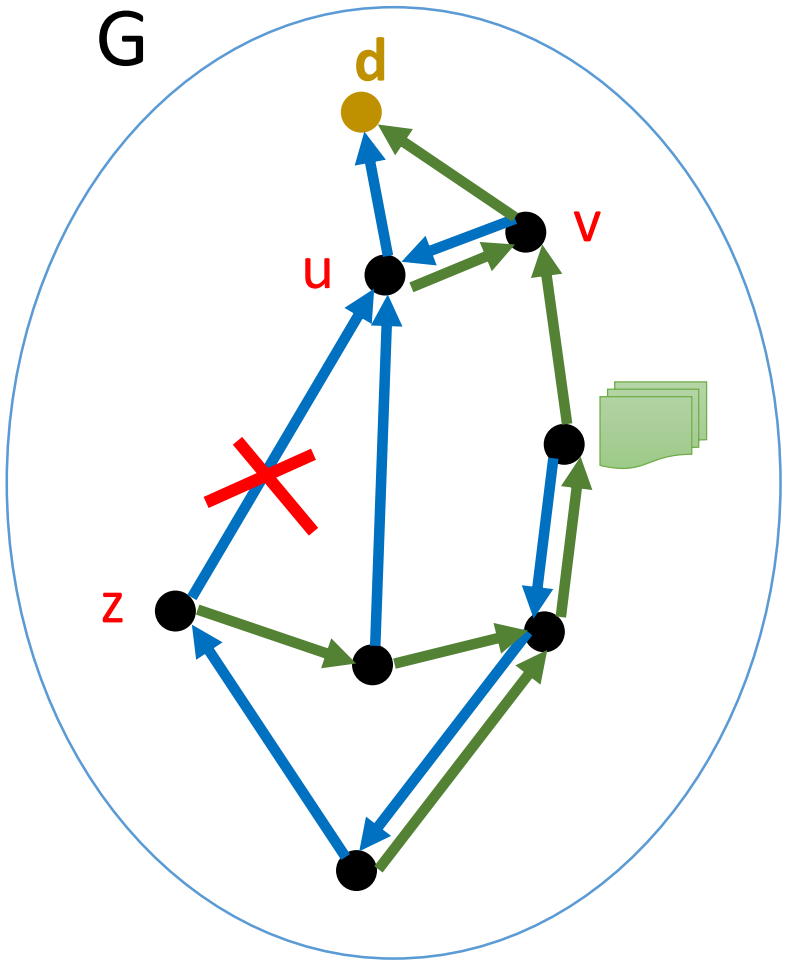


[EGR-INFOCOM14]
[CNMPGSS-INFOCOM16]

Attempt 2 – Packing arborescences

[*Edmonds, 1973*] A k-edge-connected graph G contains k arc-disjoint d-rooted arborescences.

1. Find arborescences packing
2. Route along arborescences
3. On failed edge, choose the next available arborescence (*circular routing*)



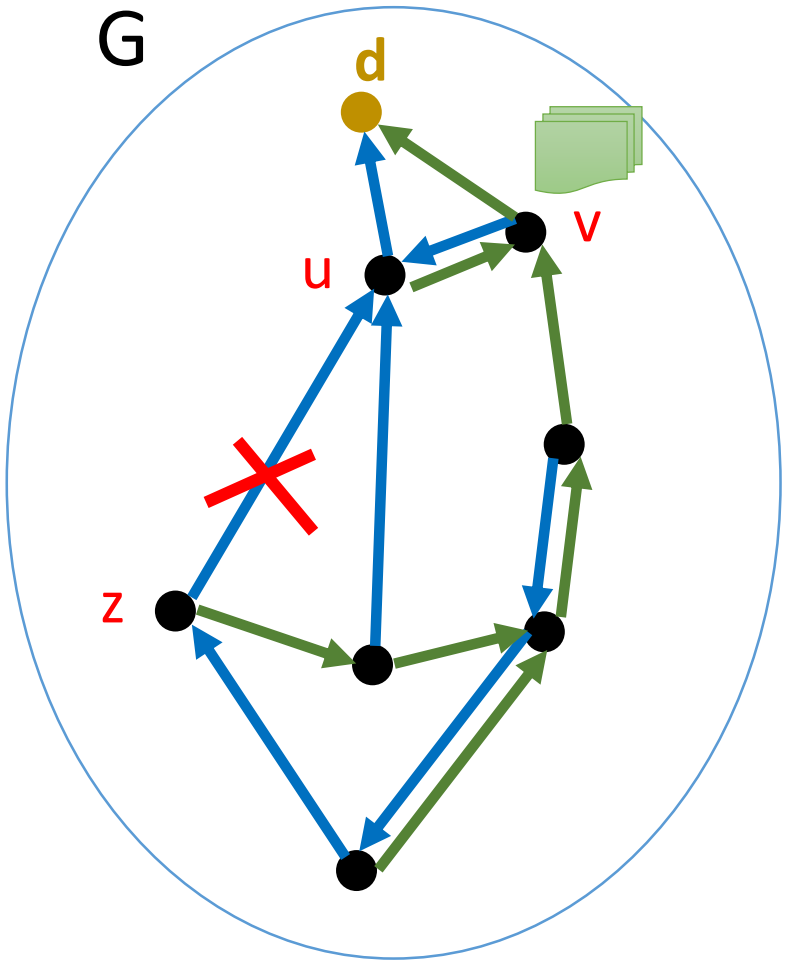
[EGR-INFOCOM14]
[CNMPGSS-INFOCOM16]

Attempt 2 – Packing arborescences

[*Edmonds, 1973*] A k-edge-connected graph G contains k arc-disjoint d-rooted arborescences.

1. Find arborescences packing
2. Route along arborescences
3. On failed edge, choose the next available arborescence (*circular routing*)

[EGR-INFOCOM14]
[CNMPGSS-INFOCOM16]

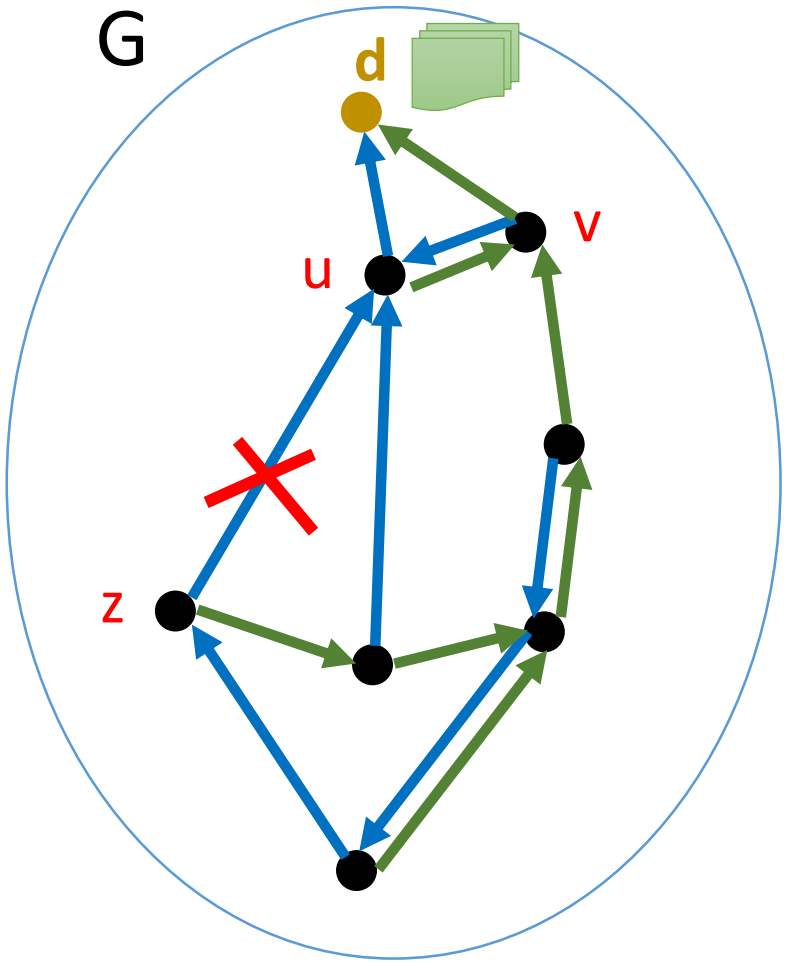


Attempt 2 – Packing arborescences

[*Edmonds, 1973*] A k-edge-connected graph G contains k arc-disjoint d-rooted arborescences.

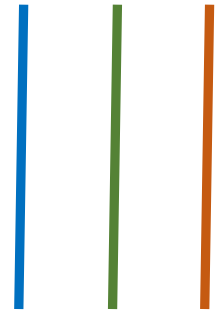
1. Find arborescences packing
2. Route along arborescences
3. On failed edge, choose the next available arborescence (*circular routing*)

[EGR-INFOCOM14]
[CNMPGSS-INFOCOM16]

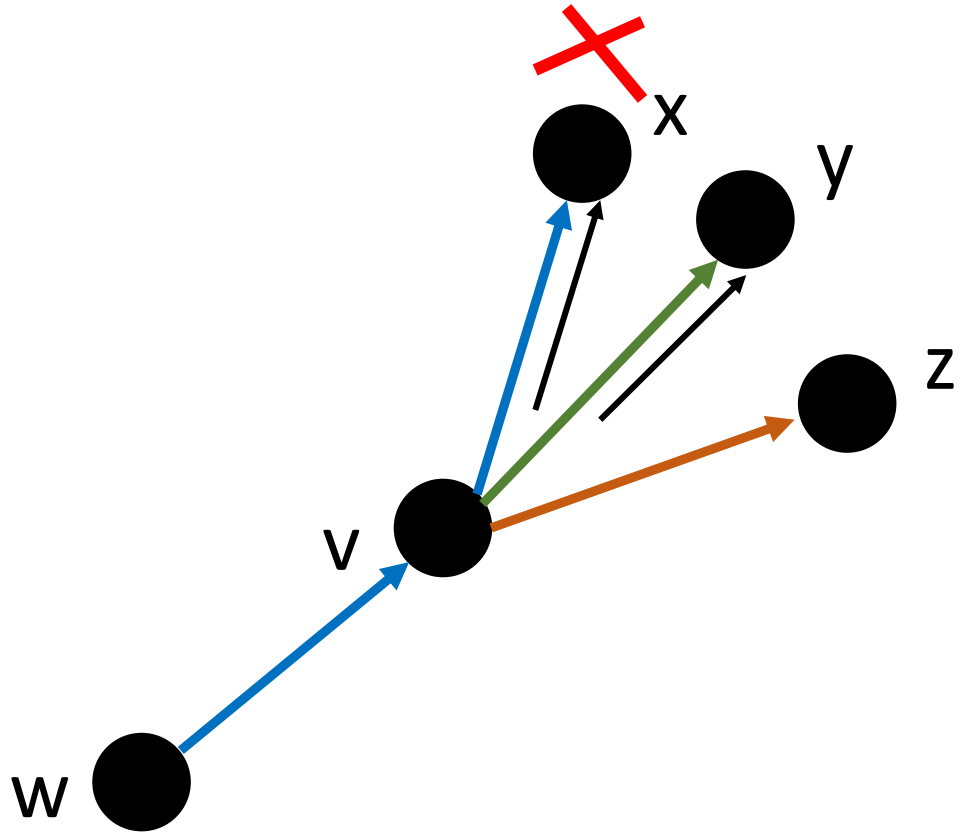


Arborescences give k/2 resilience

- Find an arborescence packing
- Order the arborescences



- Route along arborescences; on failed edge route along the next arborescence in the ordering



Each failure can affect two arborescences.

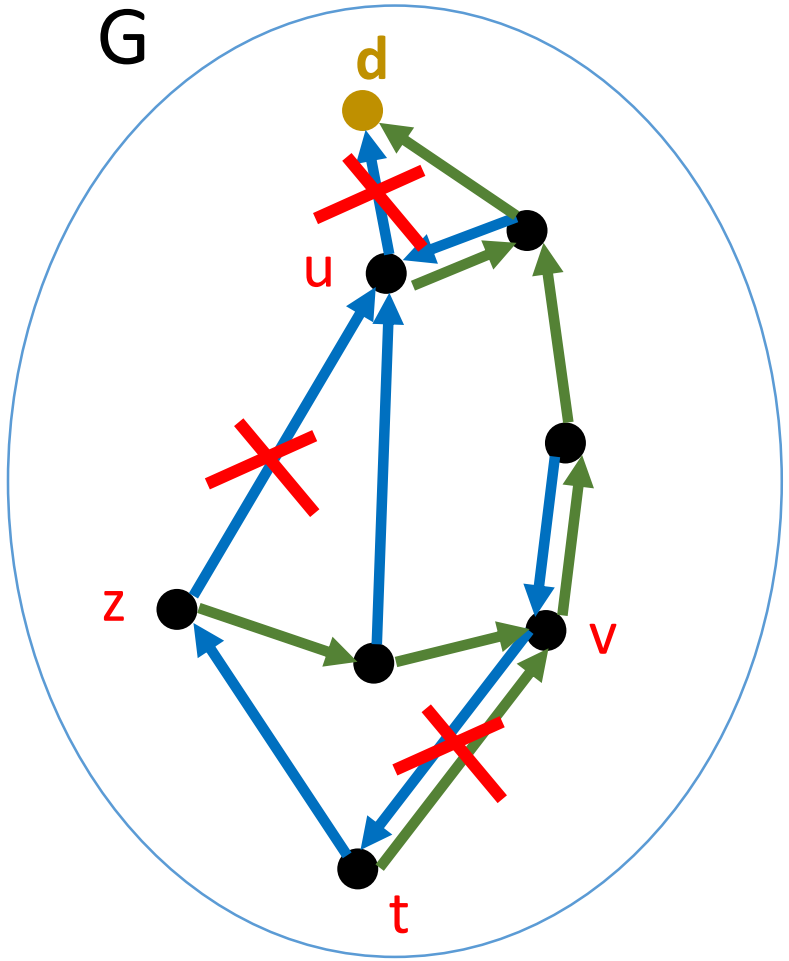
The example is tight (think of a cycle).

Can we do better?

Our result: We settle this challenge and, given a *k*-connected graph, provide a randomized algorithm that is *(k-1)*-resilient.

(Naive use of randomization would be to just take a random walk. But this would be very inefficient both in amount of randomness used and the expected length of the routing paths.)

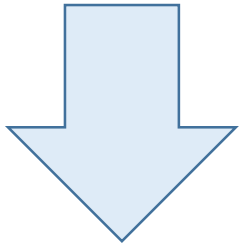
Two types of failed edges



- {t, v} edge is shared
- edges {z, u} and {u, d} are non-shared

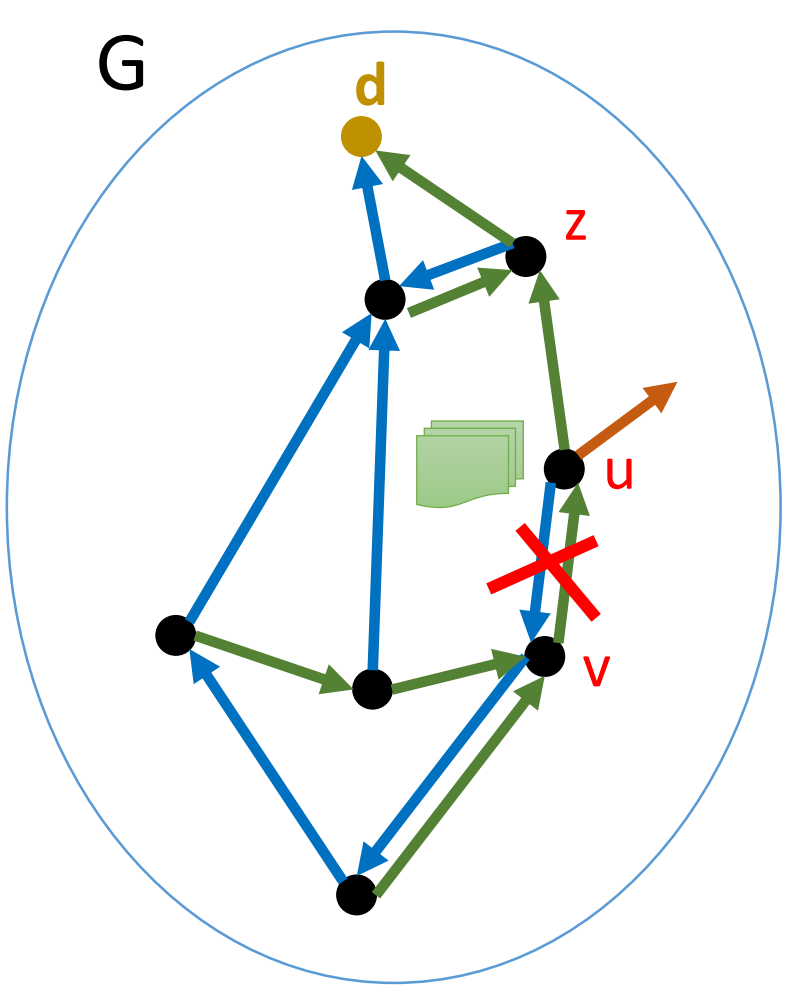
Non-shared failed edges are not a problem

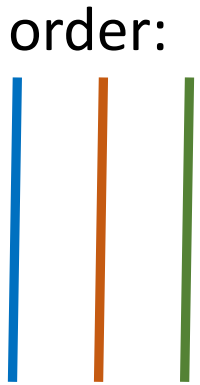
One failed edge destroys at most one arborescence



As there are at most k-1 failed edges, at least one arborescence has no failed arc.

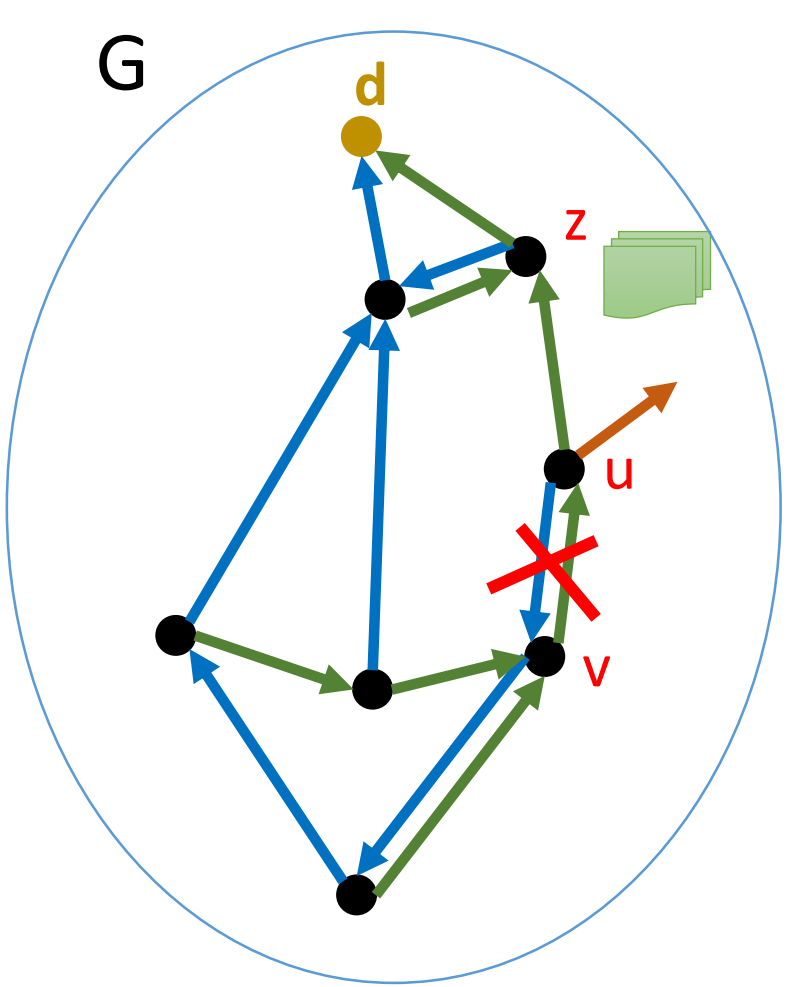
How to "recycle" failed edges? Bounce!

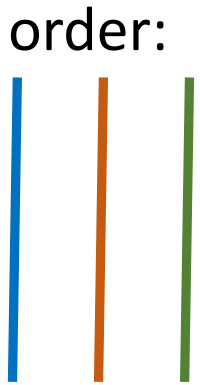




- Route from u along blue.
- If (u, v) failed, no blue u-**d** path, but ...
- (u, v) is shared and there is a green u-**d** path!
- So, bounce at u.

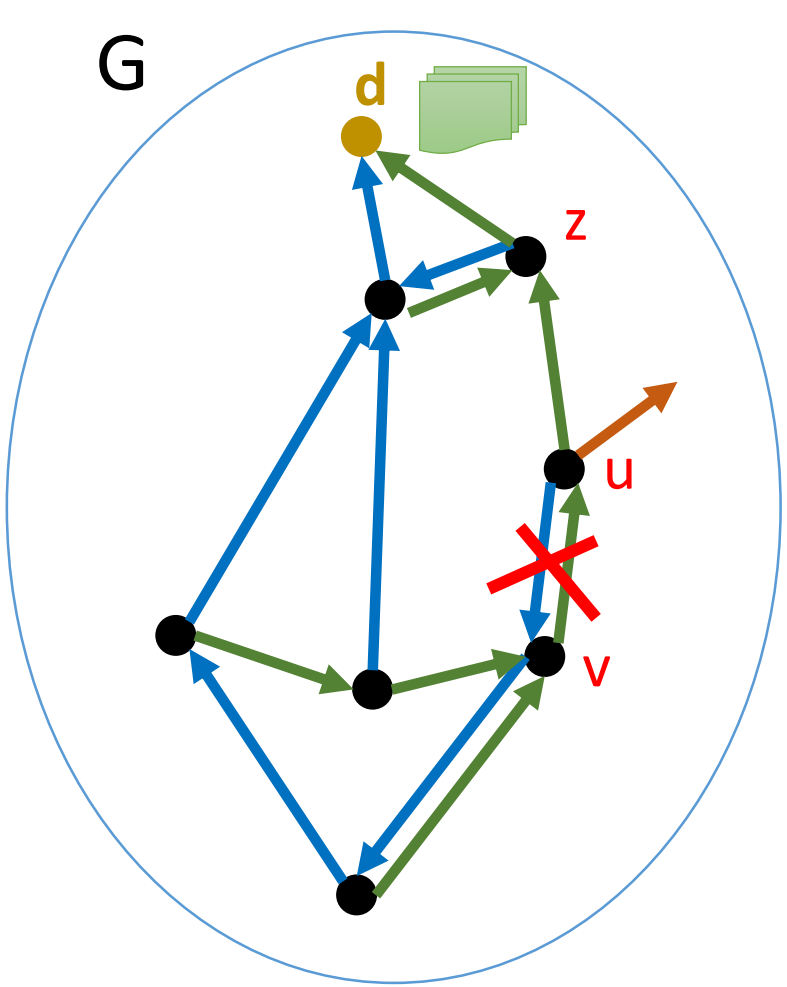
How to "recycle" failed edges? Bounce!

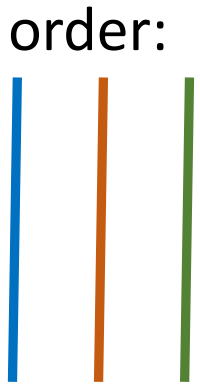




- Route from u along blue.
- If (u, v) failed, no blue u-**d** path, but ...
- (u, v) is shared and there is a green u-**d** path!
- So, bounce at u.

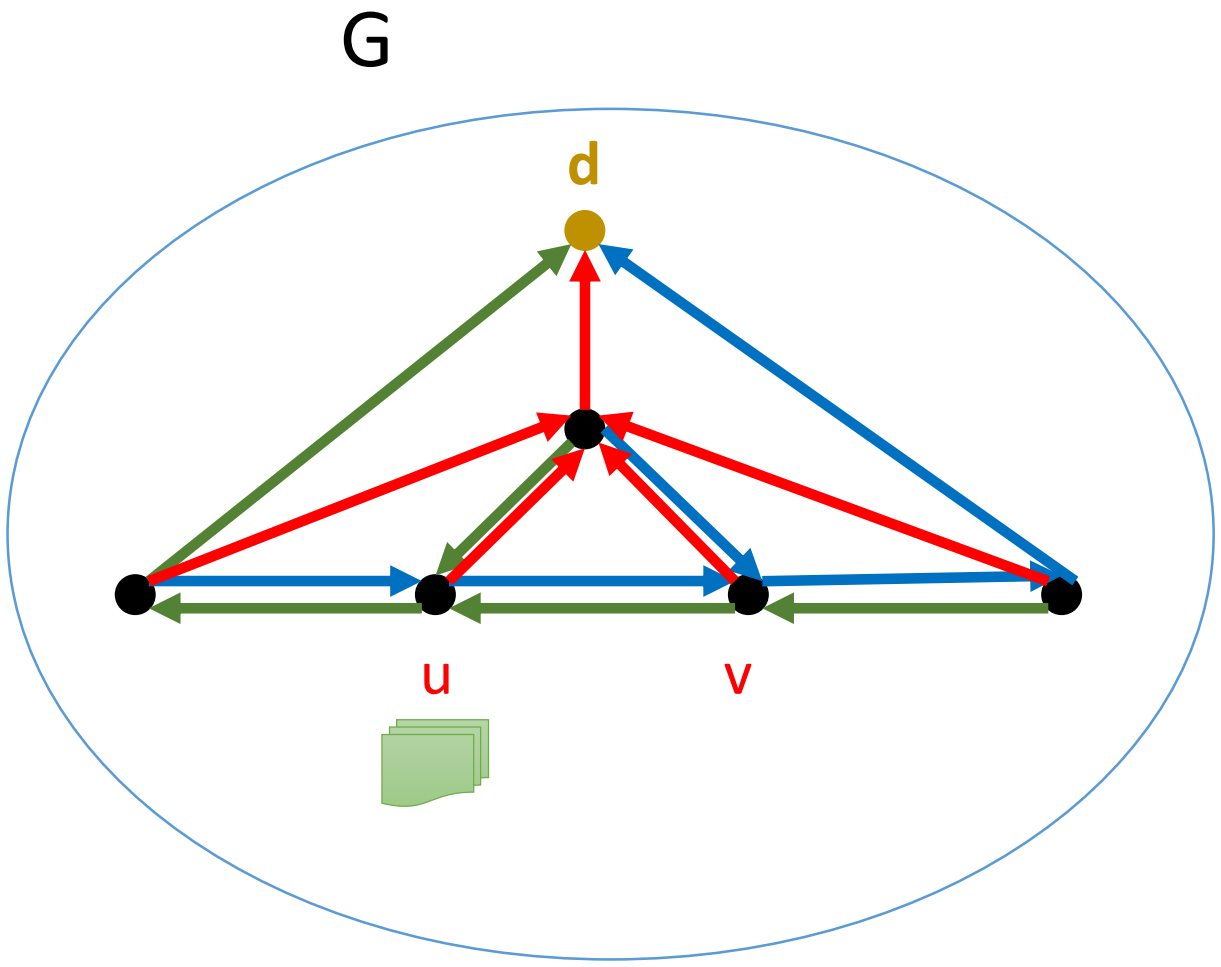
How to "recycle" failed edges? Bounce!





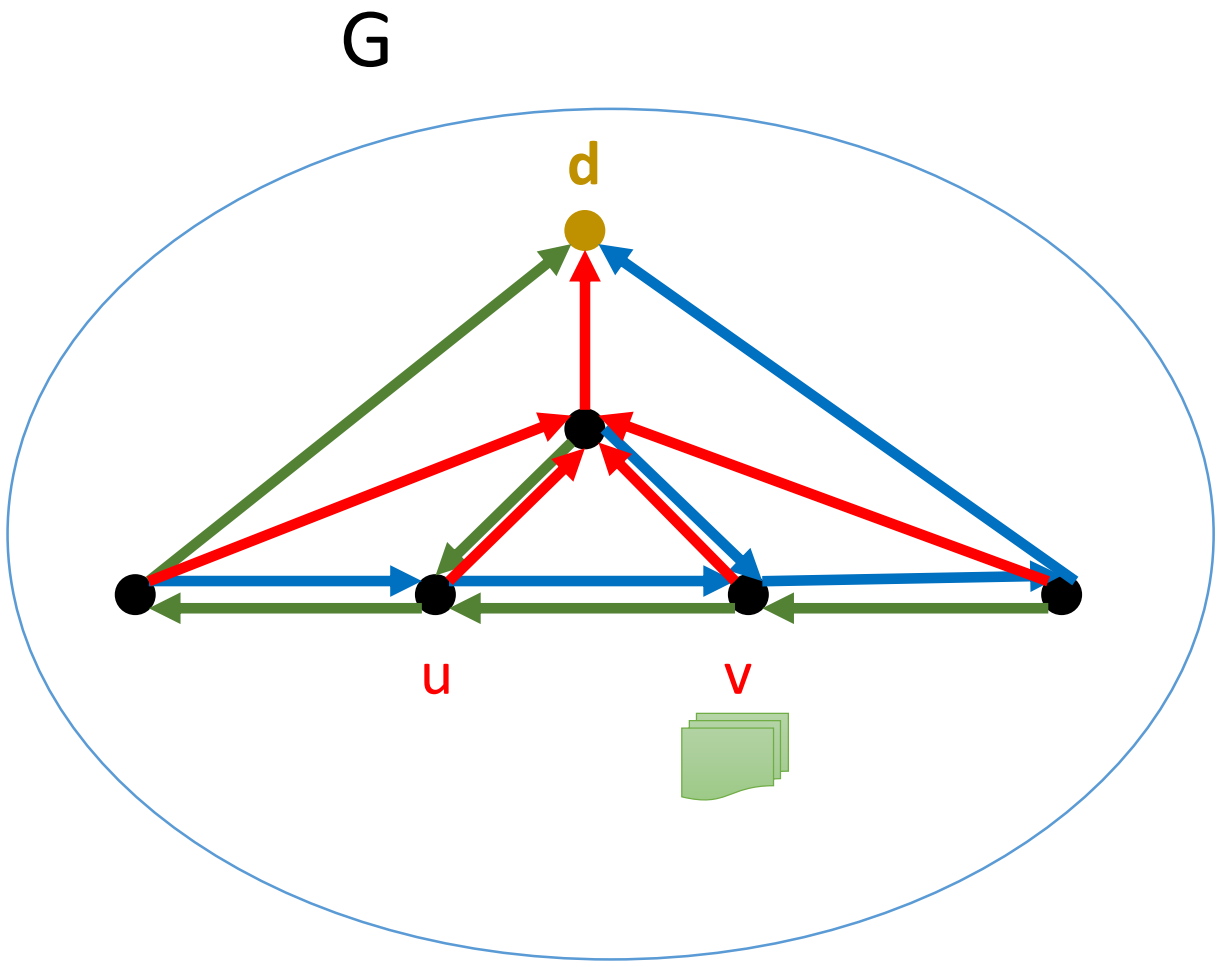
- Route from u along blue.
- If (u, v) failed, no blue u-**d** path, but ...
- (u, v) is shared and there is a green u-**d** path!
- So, bounce at u.

Bouncing every time does not work



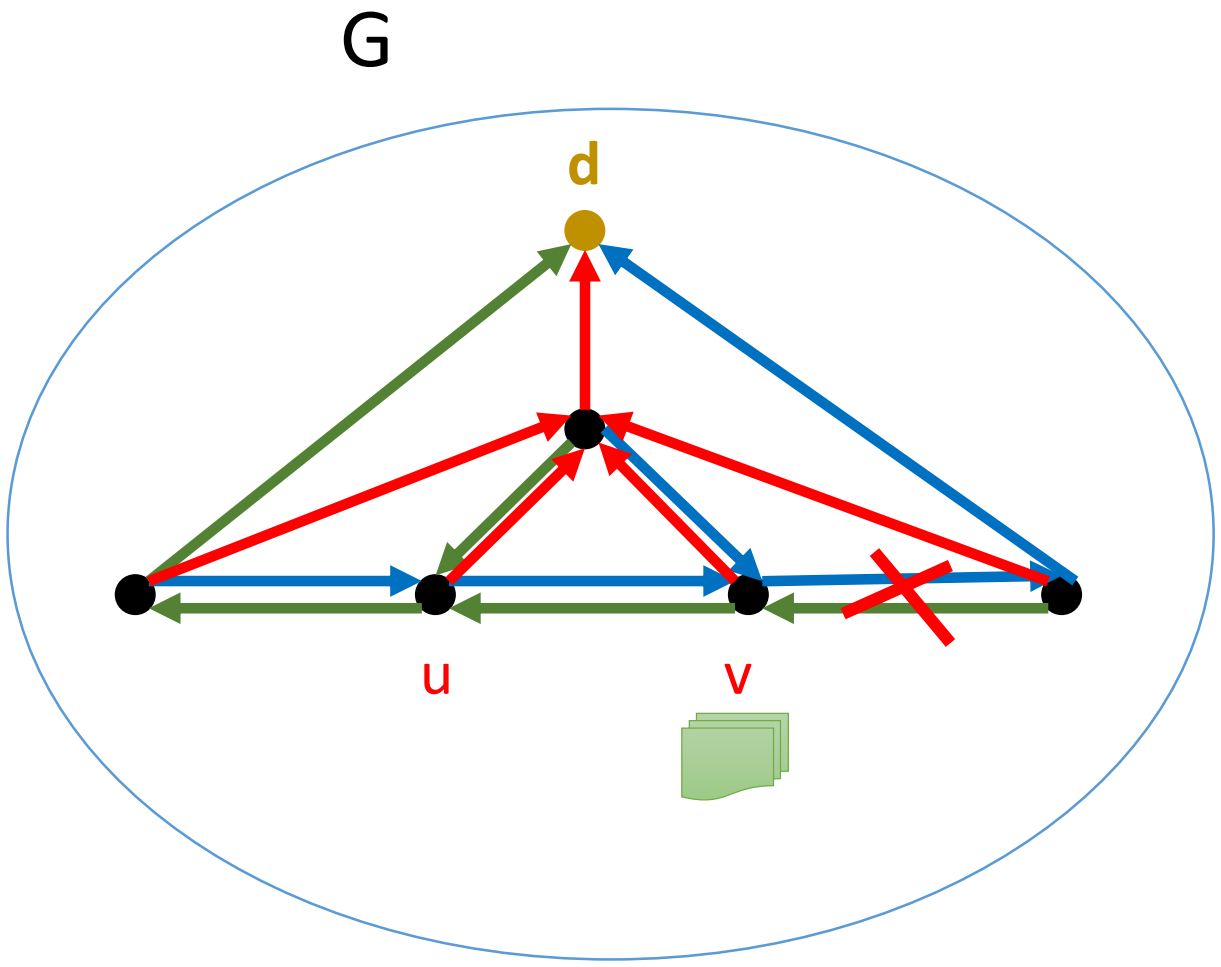
1. The packet is on blue at u.

Bouncing every time does not work



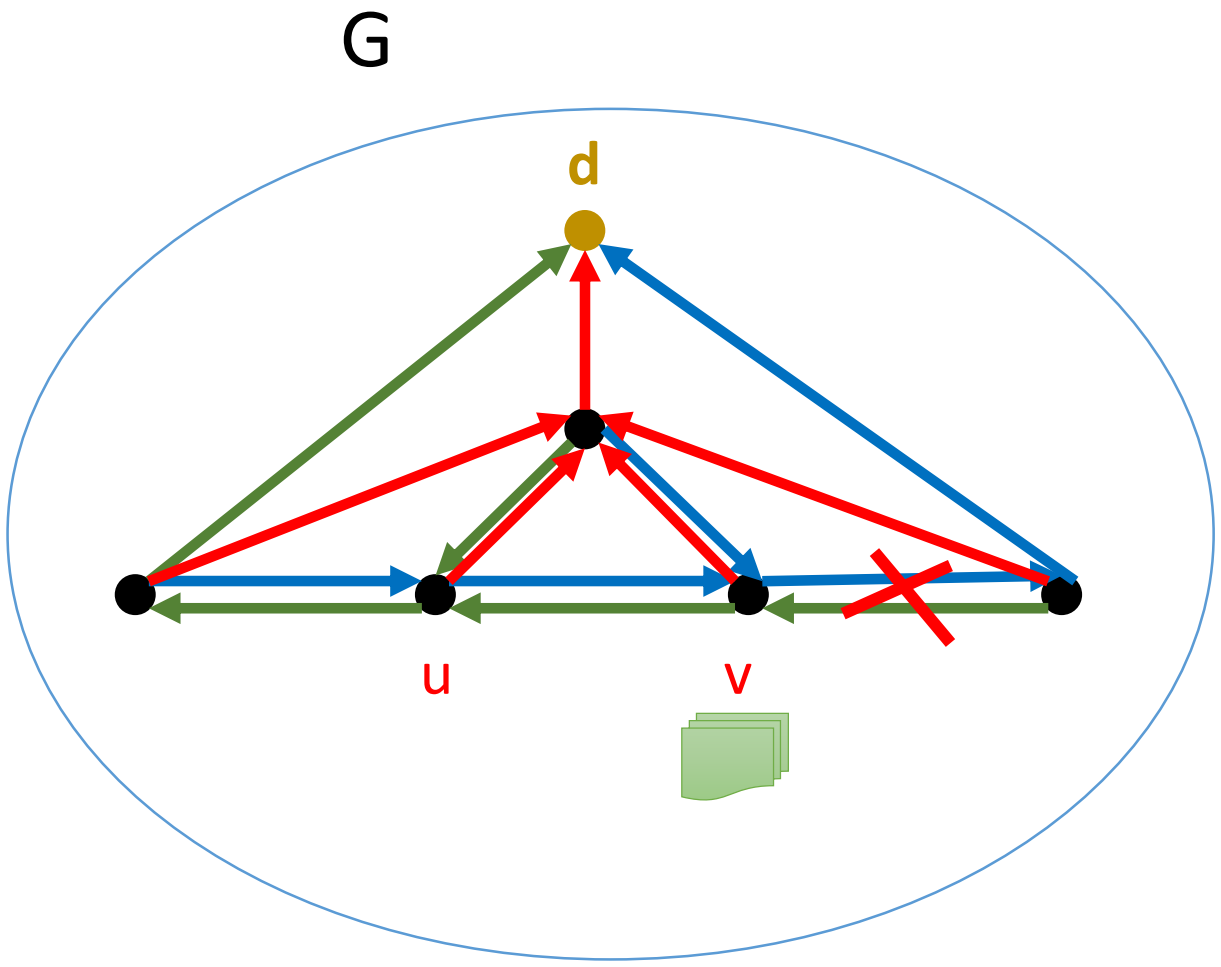
1. The packet is on blue at u.
2. The packet is on blue at v.

Bouncing every time does not work



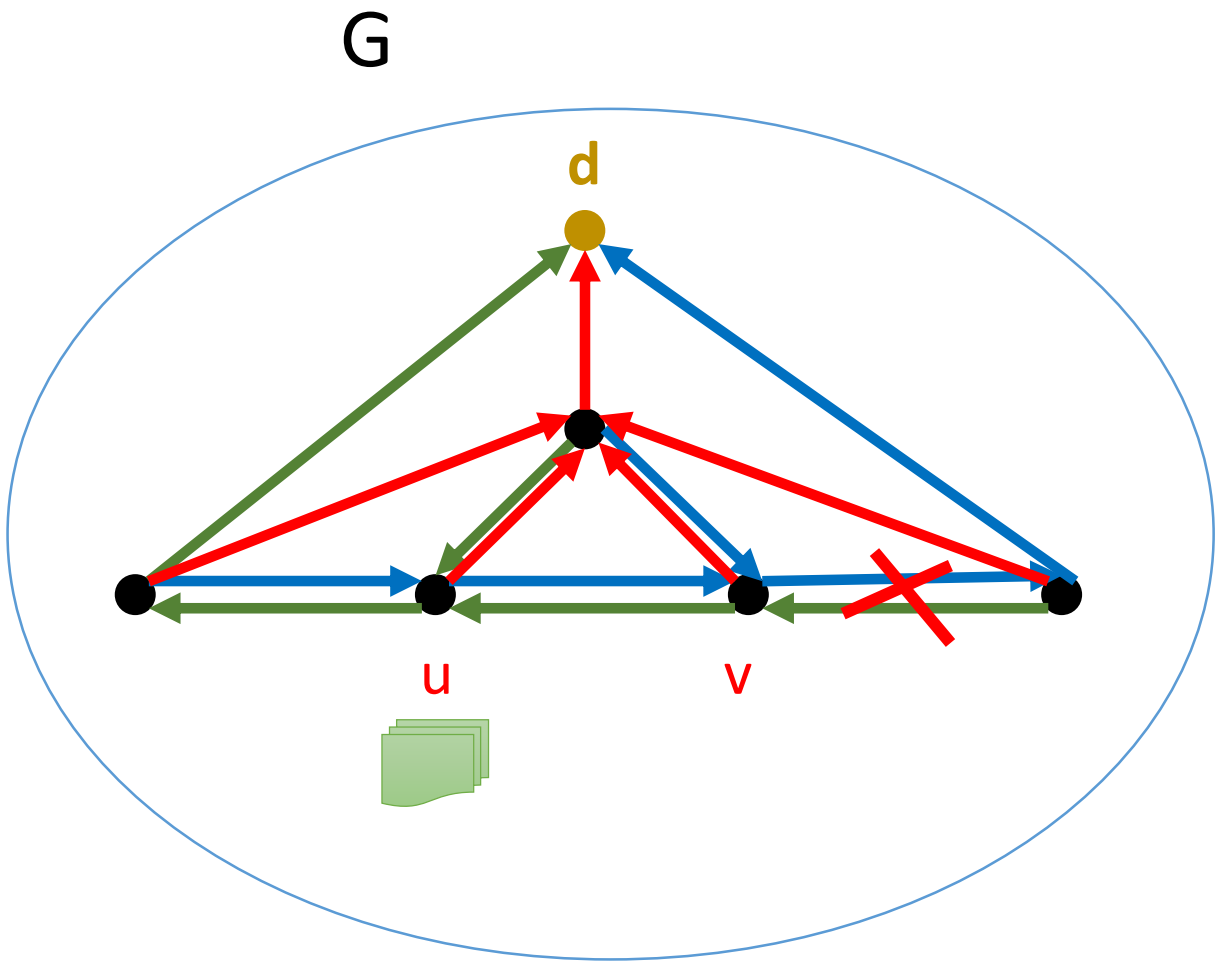
1. The packet is on blue at u.
2. The packet is on blue at v.
3. The next link failed. Bounce.

Bouncing every time does not work



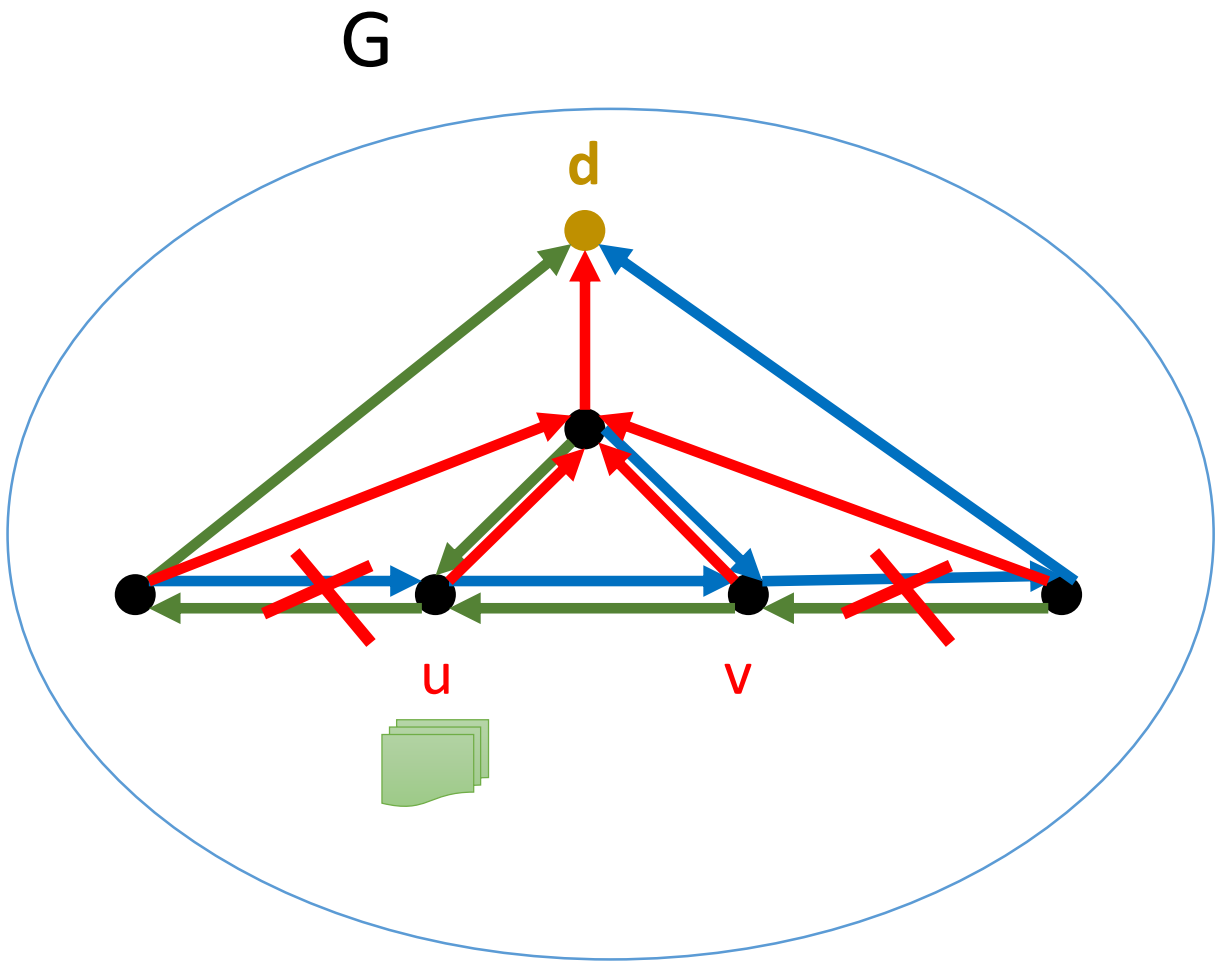
1. The packet is on blue at u.
2. The packet is on blue at v.
3. The next link failed. Bounce.
4. The packet is on green at v.

Bouncing every time does not work



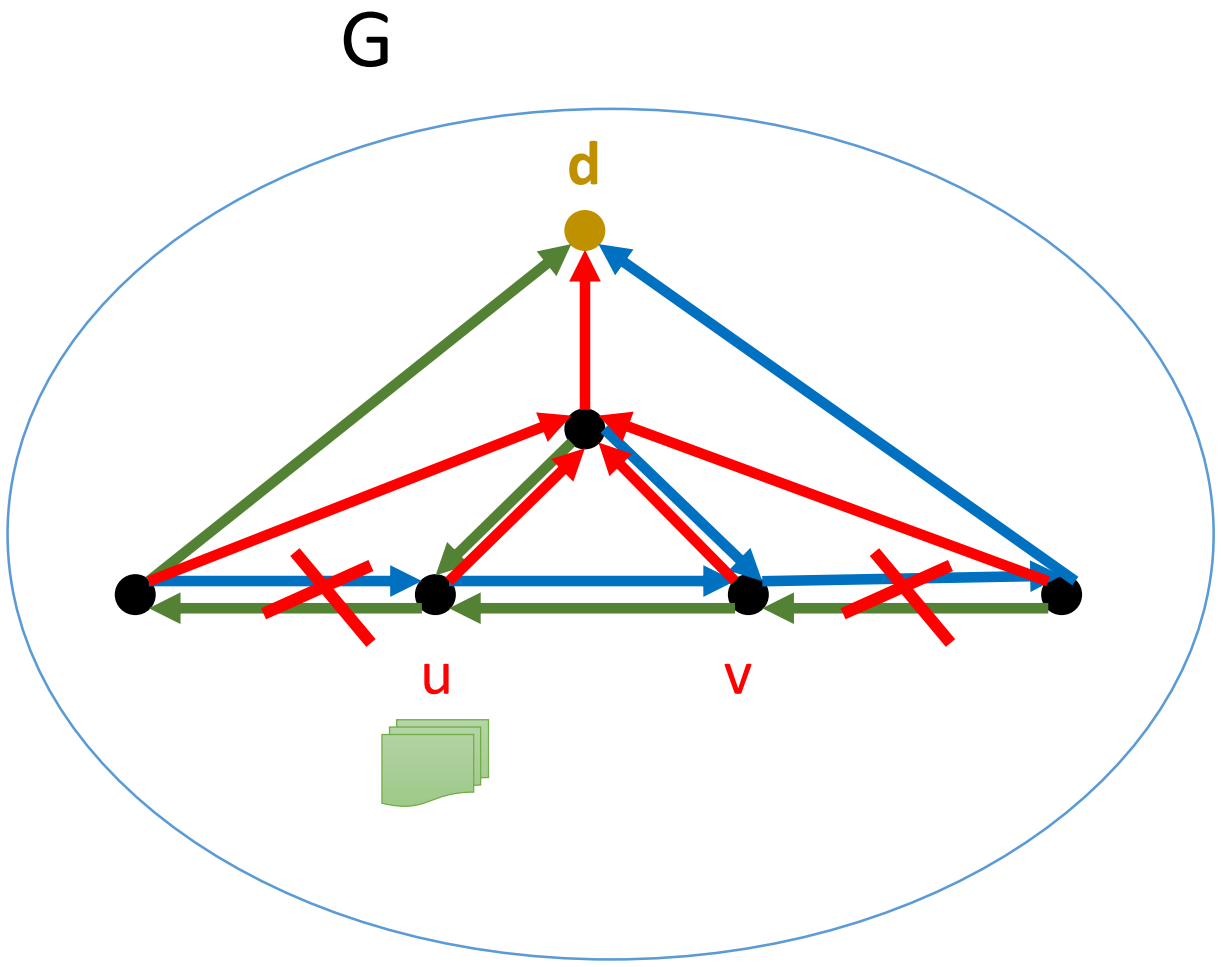
1. The packet is on blue at u.
2. The packet is on blue at v.
3. The next link failed. Bounce.
4. The packet is on green at v.
5. The packet is on green at u.

Bouncing every time does not work



1. The packet is on blue at u.
2. The packet is on blue at v.
3. The next link failed. Bounce.
4. The packet is on green at v.
5. The packet is on green at u.
6. The next link failed. Bounce.

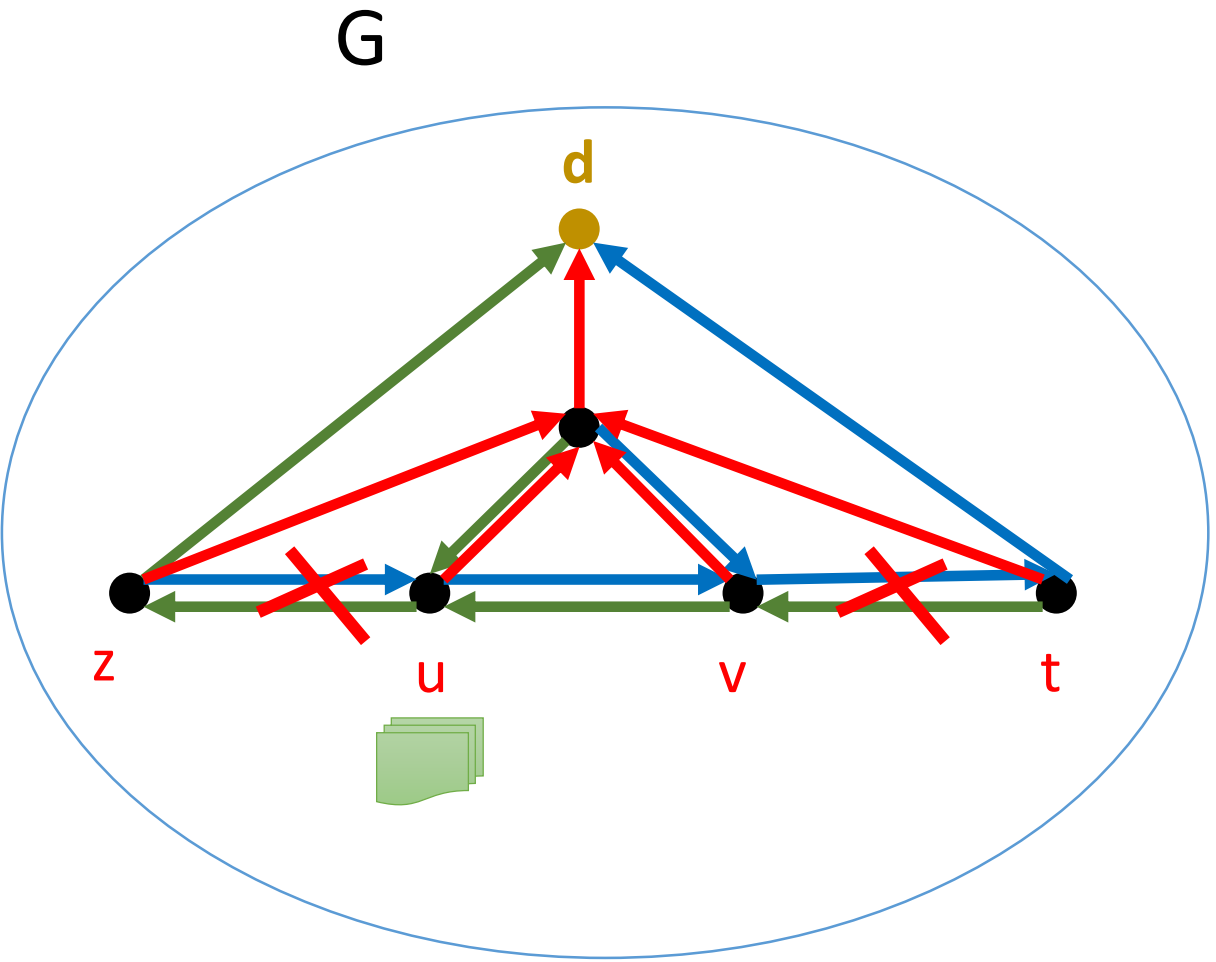
Bouncing every time does not work



1. The packet is on blue at u.
2. The packet is on blue at v.
3. The next link failed. Bounce.
4. The packet is on green at v.
5. The packet is on green at u.
6. The next link failed. Bounce.
7. The packet is on blue at u.

LOOP!

Well-bouncing arcs and good arborescences

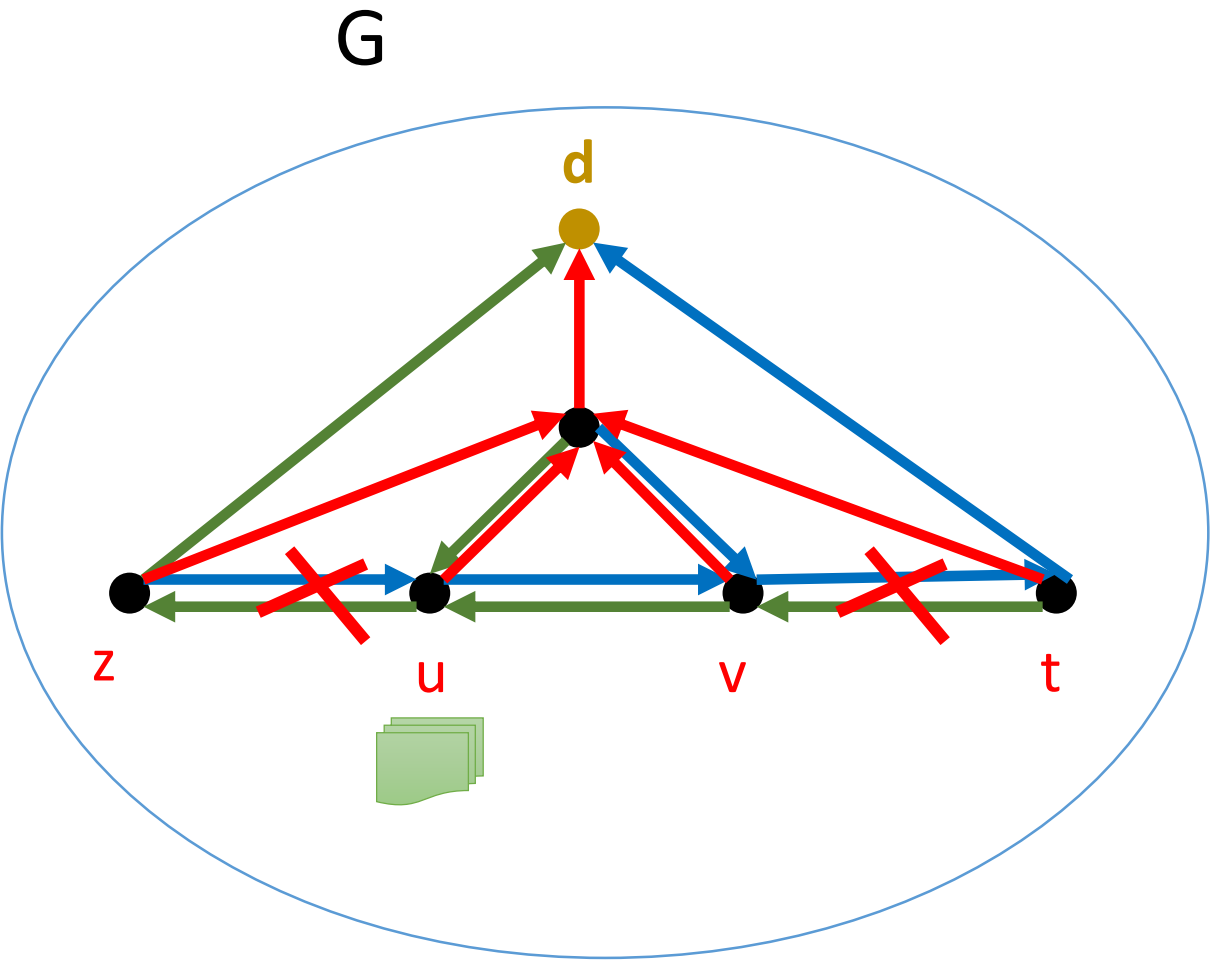


- An arc is *well-bouncing arcs* if bouncing on it takes a packet to **d** without any interruption. (<u>no more loops</u>)
- An arborescence is *good* if its every failed arc is well-bouncing.

(u, z) and (v, t) are <u>not</u> well-bouncing

(z, u) and (t, v) are well bouncing

Well-bouncing arcs and good arborescences



- An arc is *well-bouncing arcs* if bouncing on it takes a packet to **d** without any interruption. (no more loops)
- An arborescence is *good* if its every failed arc is well-bouncing.

We can show that such good arborescence always exists!

Goal now: identify good arborescence

When to bounce?

If we know the given arborescence is good, we can bounce.

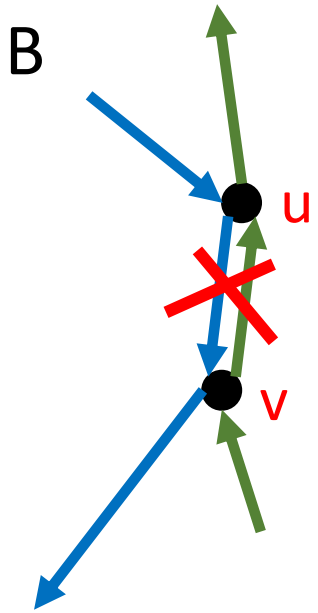
- Route along arborescence B
- If a packet hits a failed link:
  - If B is not good, route circularly
  - Otherwise, bounce

Too good to be true ... how do we know if an arborescence is good?

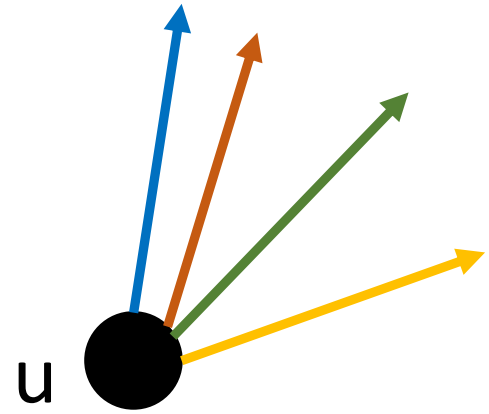
We don’t. We guess!

Goal: routing on a good arborescence

- Route along arborescence B
- If a packet hits a failed link make a guess



With prob *p*



With prob 1-*p*

(*p* is our likelihood that B is good)

Our main result

Theorem:

Let G be a k-connected graph, and A be a decomposition of G into k arc-disjoint arborescences rooted at **d**. Assume that there are at most f many failed edges. Then, A contains at least k-f good arborescenes.

(for details see our paper)

Efficiency of our algorithm

- *(k-1)*-resilient routing scheme using randomness.
- Number of failed links a packet hits is $\Theta\left(\frac{k}{k-f}\right)$ by choosing appropriate *p*. (more details in the paper)
- If $f < (1-\varepsilon)k$, for constant $\varepsilon < 1$, the packet hits only a constant number of failed links in expectation.

Open problems

- Can we devise deterministic (*k-1*)-resilient routing tables?
- Can we slightly alter the given network and improve the resiliency?
- Can we devise even a randomized (*k-1*)-vertex-resilient routing scheme, if the network is *k*-vertex-connected?