Faster Deterministic Worst-Case Dynamic All-Pairs Shortest Paths

Shiri Chechik Tianyi Zhang

TEL AVIV UNIVERSITY
אוניברסיטת תל אביב

Dynamic All-Pairs Shortest Paths

- Given a weighted digraph $G = (V, E, \omega)$
- A sequence of **vertex updates**, maintain **pairwise exact distances**

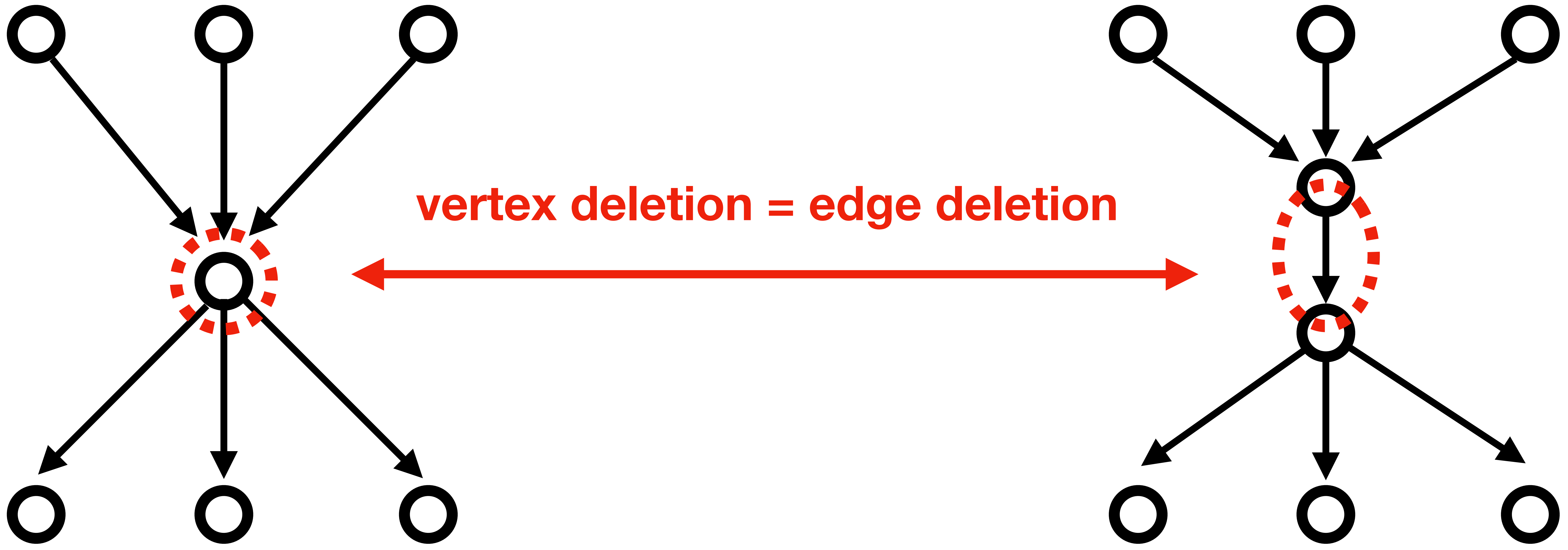
More specifically, want a data structure:

- **Ins**(v, $\mathrm{adj}(v)$) / **Del**(v)
  Insert/delete vertex v in G with adjacency list $\mathrm{adj}(v)$; **want n^2 runtime**
- **Query**(u, v)
  Return the shortest distance from u to v in G; **want $O(1)$ runtime**

Dynamic All-Pairs Shortest Paths

- Given a weighted digraph $G = (V, E, \omega)$
- A sequence of **vertex updates**, maintain **pairwise exact distances**

Why vertex updates, not edge updates?



History

reference	vertex update time	deterministic / randomized	worst-case / amortized
King, 1999	$\tilde{O}(n^{2.5}\sqrt{W})$	deterministic	amortized
Demetrescu, Italiano, 2004	$\tilde{O}(n^2)$	deterministic	amortized
Thorup, 2005	$\tilde{O}(n^{3-1/4})$	deterministic	worst-case
Abraham, Chechik, Krinninger, 2017	$\tilde{O}(n^{3-1/3})$	randomized	worst-case
Probst, Wulff-Nilsen, 2020	$\tilde{O}(n^{3-2/7})$	deterministic	worst-case
New	$\tilde{O}(n^{3-20/61})$	deterministic	worst-case

n is the number of vertices in the graph, *W* refers to the maximum edge weight

Previous approaches

Reduction to batch deletion

Batch deletion data structure:

- **Prep**(G)
  Preprocess the graph G and be ready for **one batch deletion** and queries
- **Batch**(B)
  Remove a subset $B \subseteq V$ of vertices from graph G
- **Query**(u, v)
  Return the shortest distance from u to v in $G \backslash B$ in O(1) time

Reduction to batch deletion

Theorem [Thorup, 2005]

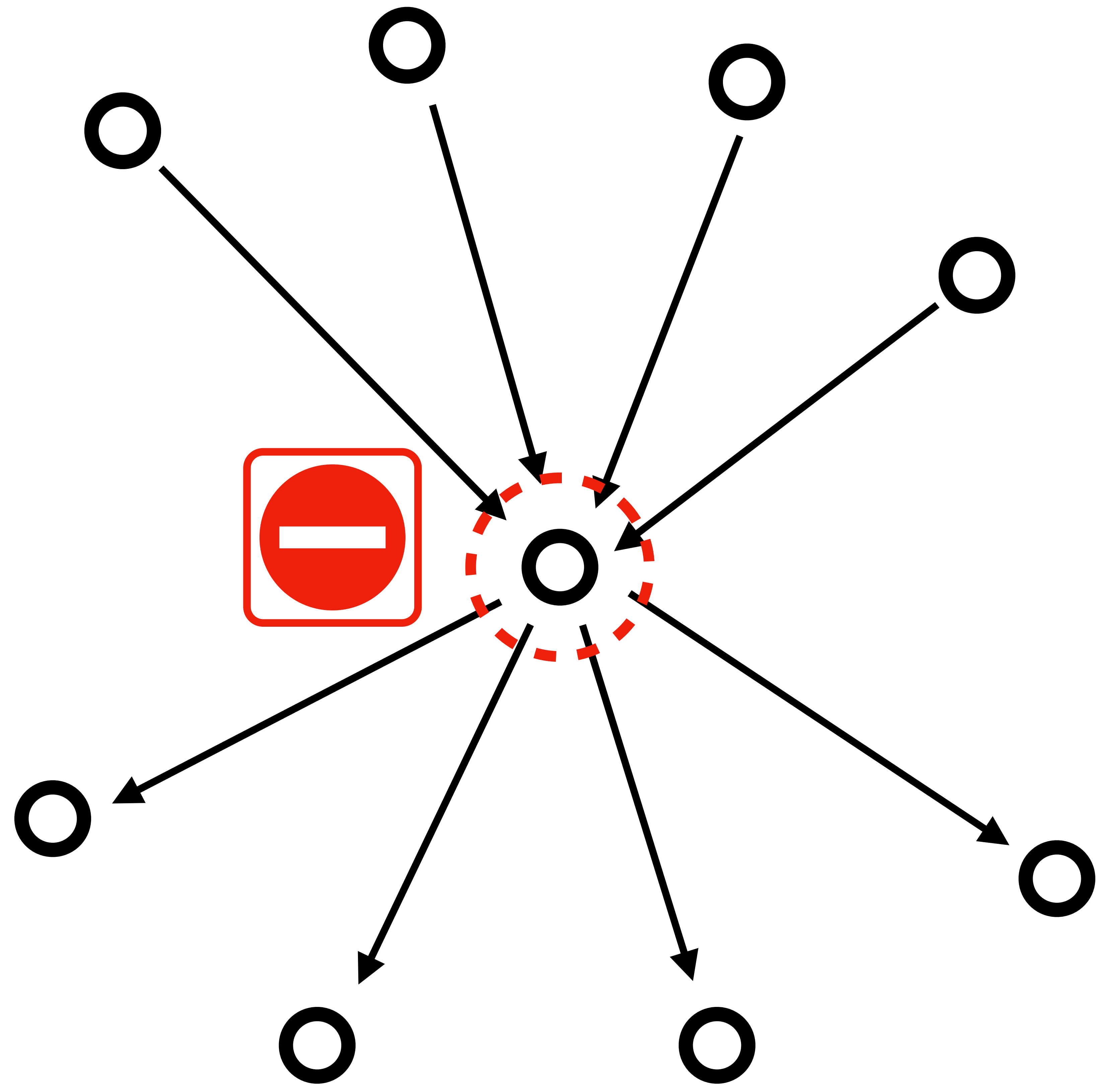
- Given a batch deletion algorithm, dynamic APSP can be solved with worst-case update time $\mathrm{T}_{\text{prep}}/|B| + \mathrm{T}_{\text{batch}} + |B|n^2$

- **Batch**(B)
  Remove a subset $B \subseteq V$ of vertices from graph G

- **Query**(u, v)
  Return the shortest distance from u to v in $G \backslash B$ in O(1) time

Batch Deletion

Main difficulty:
Precompute shortest paths in G
A single deletion can **destroy a lot**

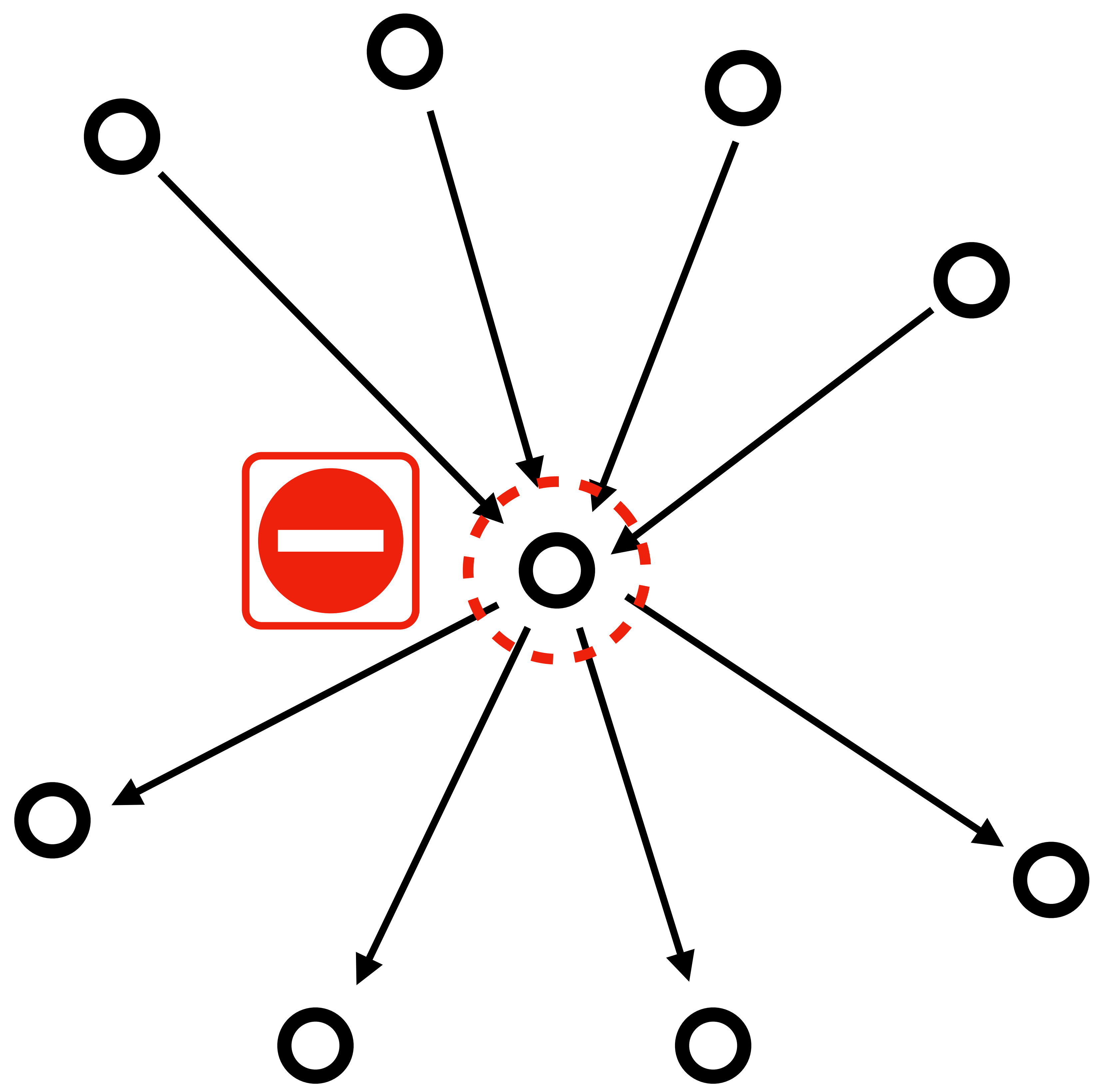


Batch Deletion

Main difficulty:
Precompute shortest paths in G
A single deletion can **destroy a lot**

Two basic ideas [Thorup, 2005]

- Shortest paths with small #hops
  Long-hop paths can be handled using hitting sets
- Prepare **low-congestion** shortest paths



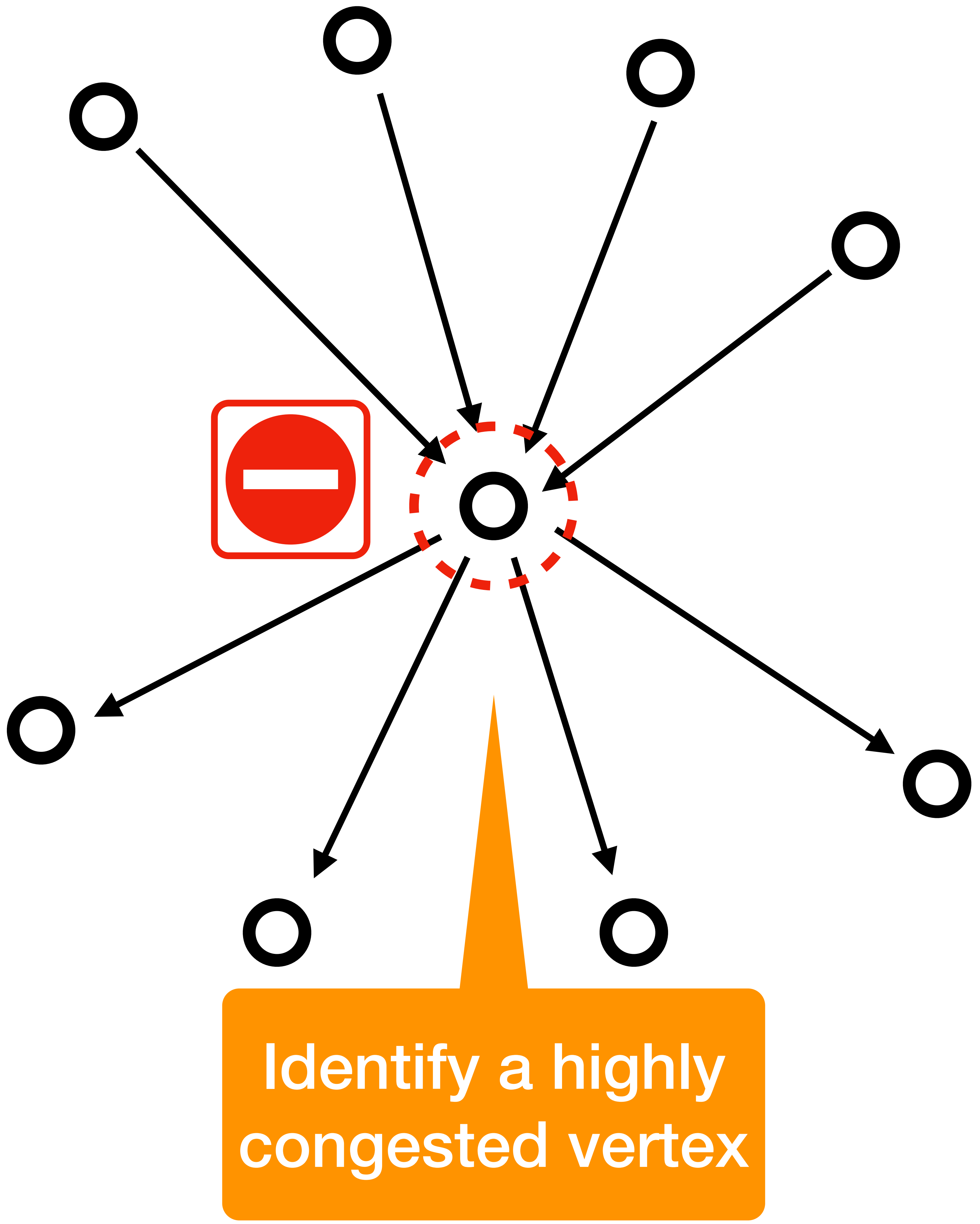
Batch Deletion

<u>Main difficulty:</u>
Precompute shortest paths in G
A single deletion can **destroy a lot**

Two basic ideas [Thorup, 2005]

- Shortest paths with small #hops
  Long-hop paths can be handled using hitting sets

- **Prepare low-congestion shortest paths**



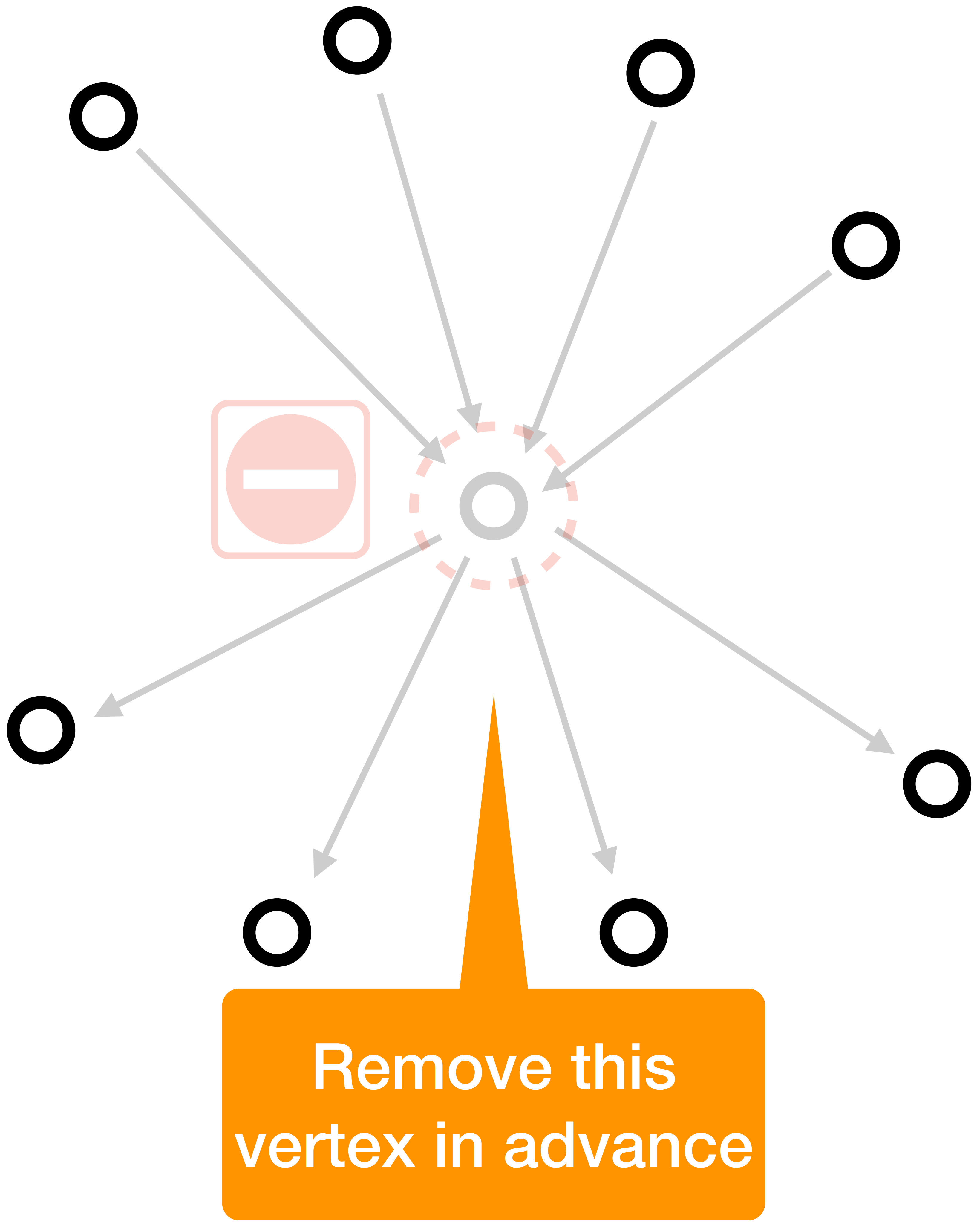
Batch Deletion

Main difficulty:
Precompute shortest paths in G
A single deletion can **destroy a lot**

Two basic ideas [Thorup, 2005]

- Shortest paths with small #hops
  Long-hop paths can be handled using hitting sets

- **Prepare low-congestion shortest paths**

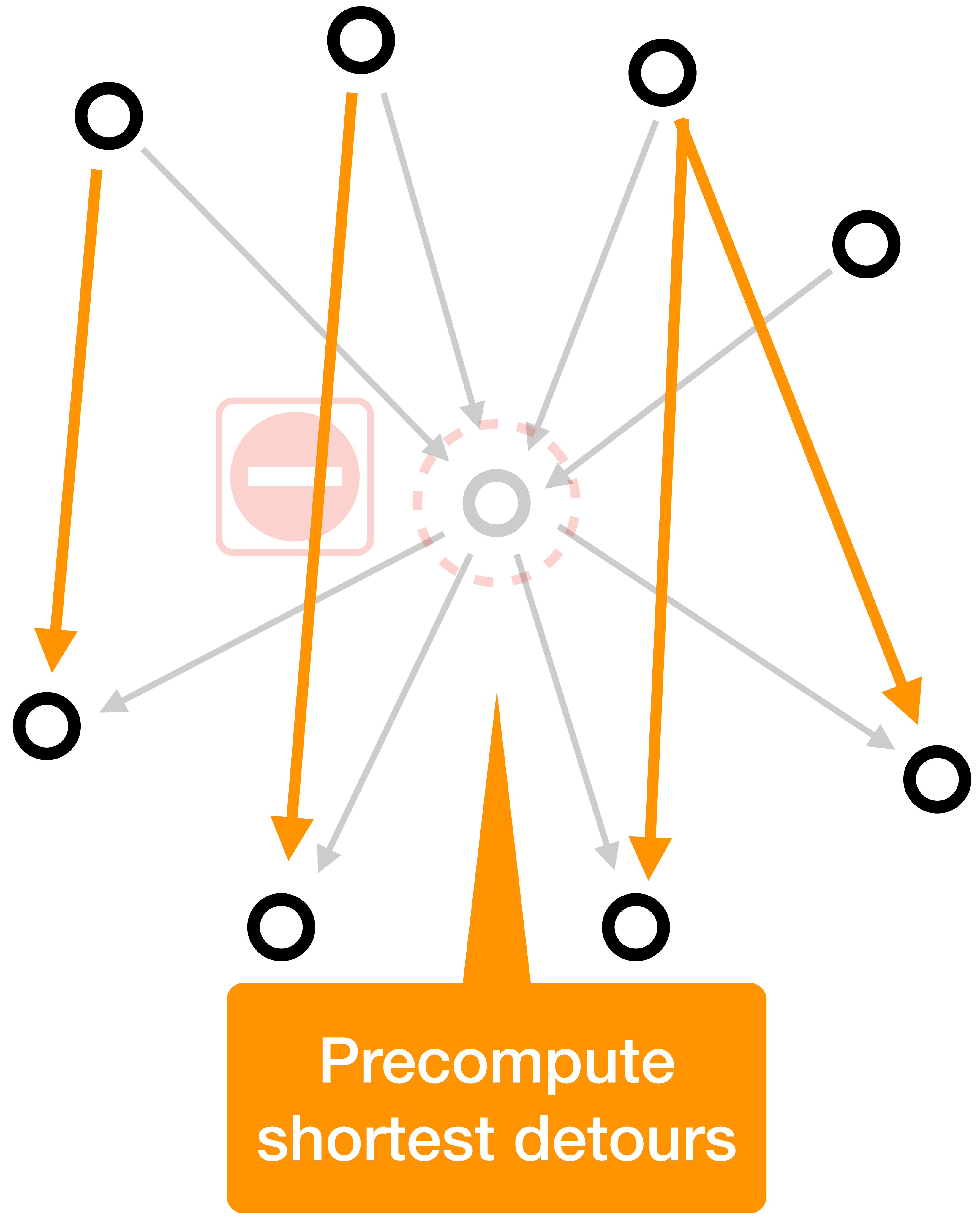


Batch Deletion

Main difficulty:
Precompute shortest paths in G
A single deletion can **destroy a lot**

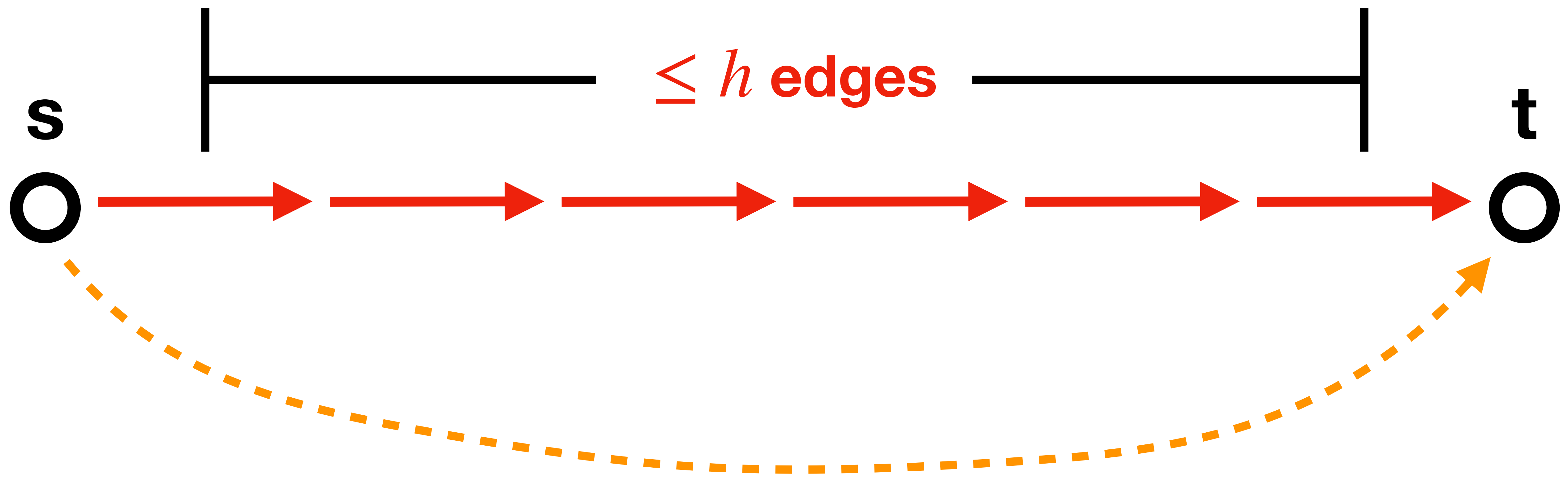
Two basic ideas [Thorup, 2005]

- Shortest paths with small #hops
  Long-hop paths can be handled using hitting sets
- **Prepare low-congestion shortest paths**



Hop-Restricted Shortest Paths

- An h-hop shortest path $\pi_{s,t}$ is the **shortest path with** **at most h edges**
- Single-source h-hop paths $\{\pi_{s,t}\}_{t \in V}$ can be computed using the **Bellman-Ford** algorithm in time $n^2 h$



Ordinary shortest path might contain $\gg h$ edges

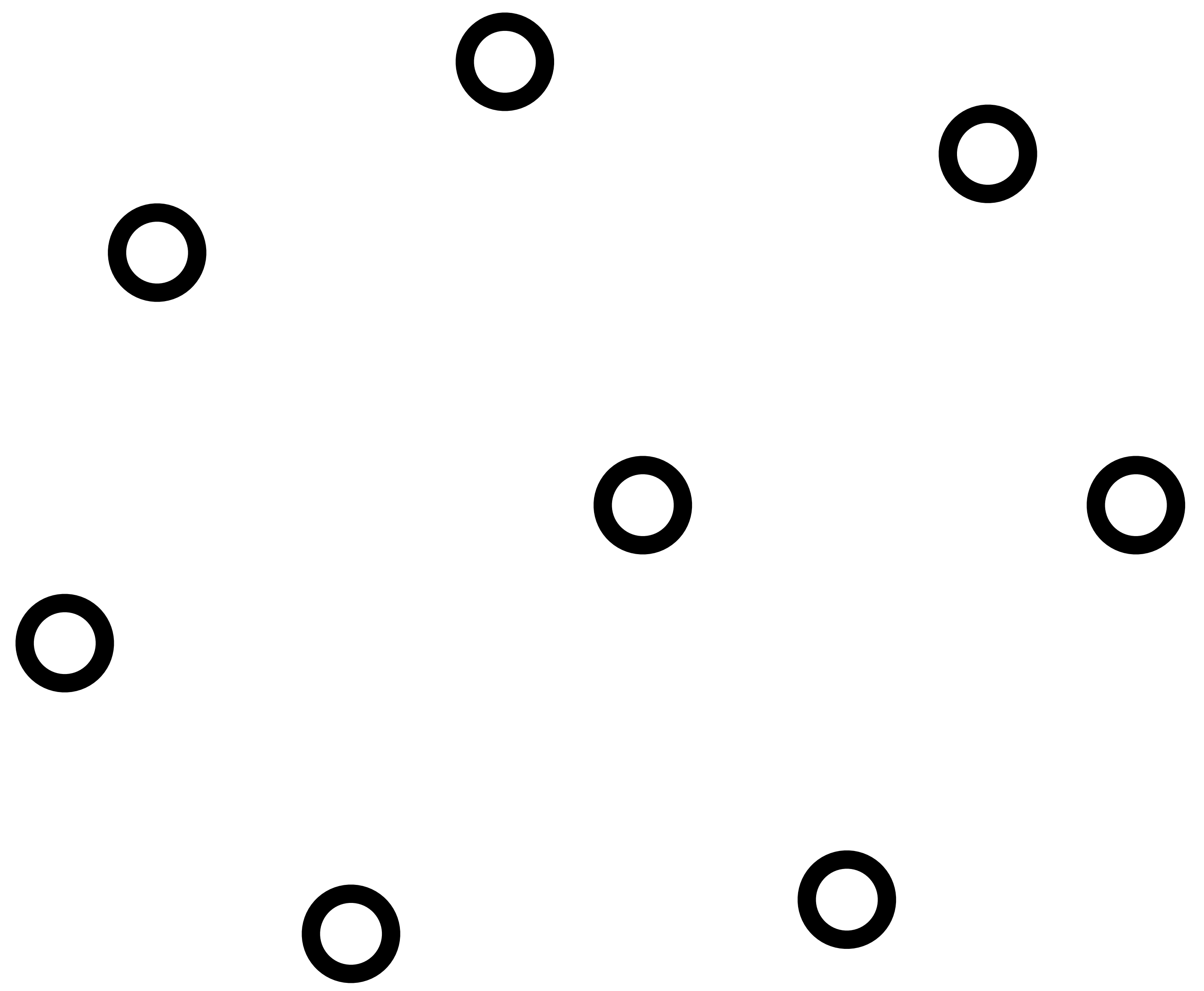
The congestion technique

Low-congestion shortest paths [Thorup'05]

1. Pick a vertex v that maximizes cg(v)
2. Compute h-hop shortest paths at v using Bellman-Ford
3. Add h-hop paths to Π, update cg(.)
4. Remove v from graph, go to Step 1

$\Pi = \{\pi_{s,t} \mid s, t \in V\}$ a set of short paths

cg(v) = #paths in Π containing v



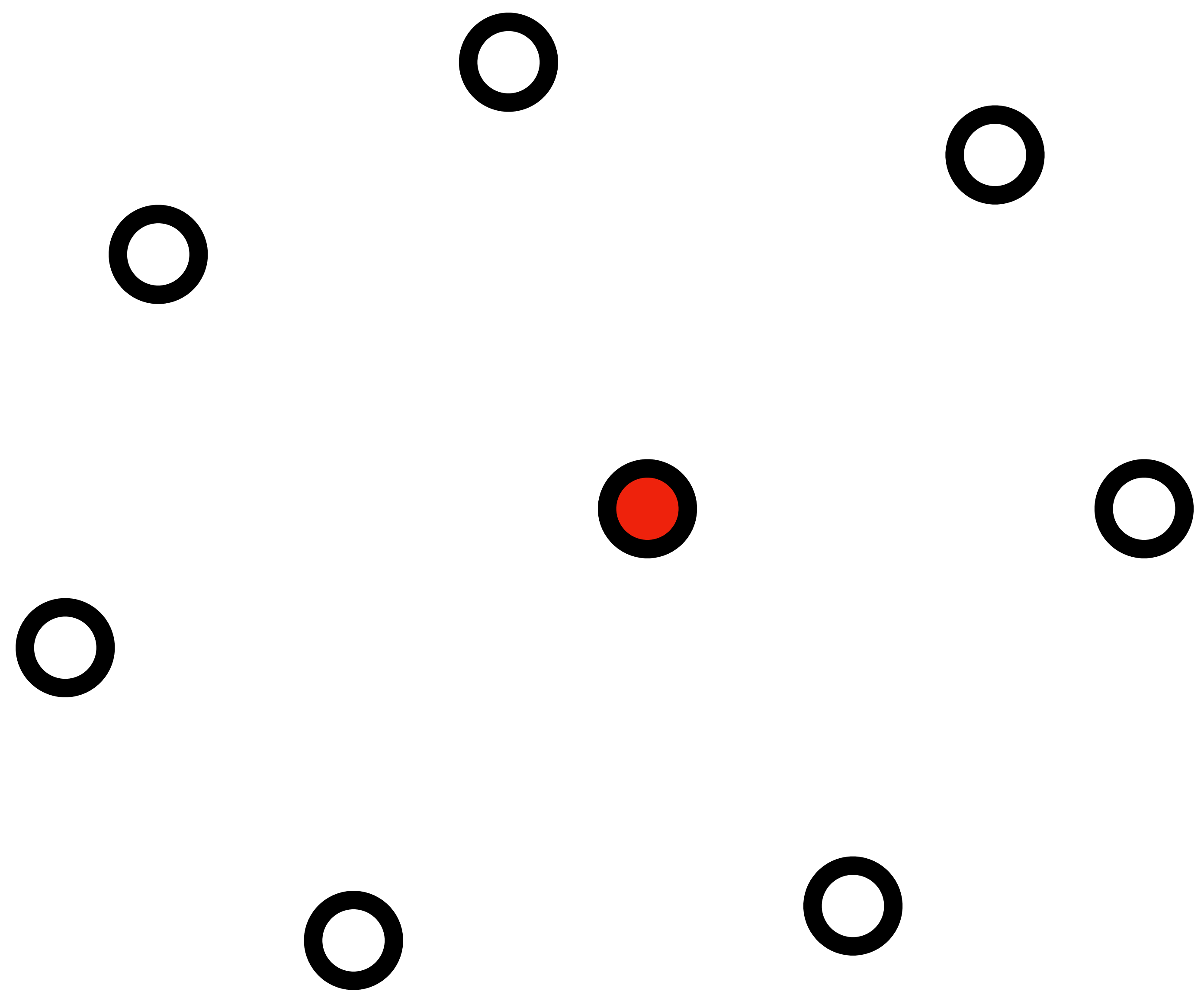
The congestion technique

Low-congestion shortest paths [Thorup'05]

1. **Pick a vertex v that maximizes cg(v)**
2. Compute h-hop shortest paths at v using Bellman-Ford
3. Add h-hop paths to Π, update cg(.)
4. Remove v from graph, go to Step 1

$\Pi = \{\pi_{s,t} \mid s, t \in V\}$ a set of short paths

cg(v) = #paths in Π containing v



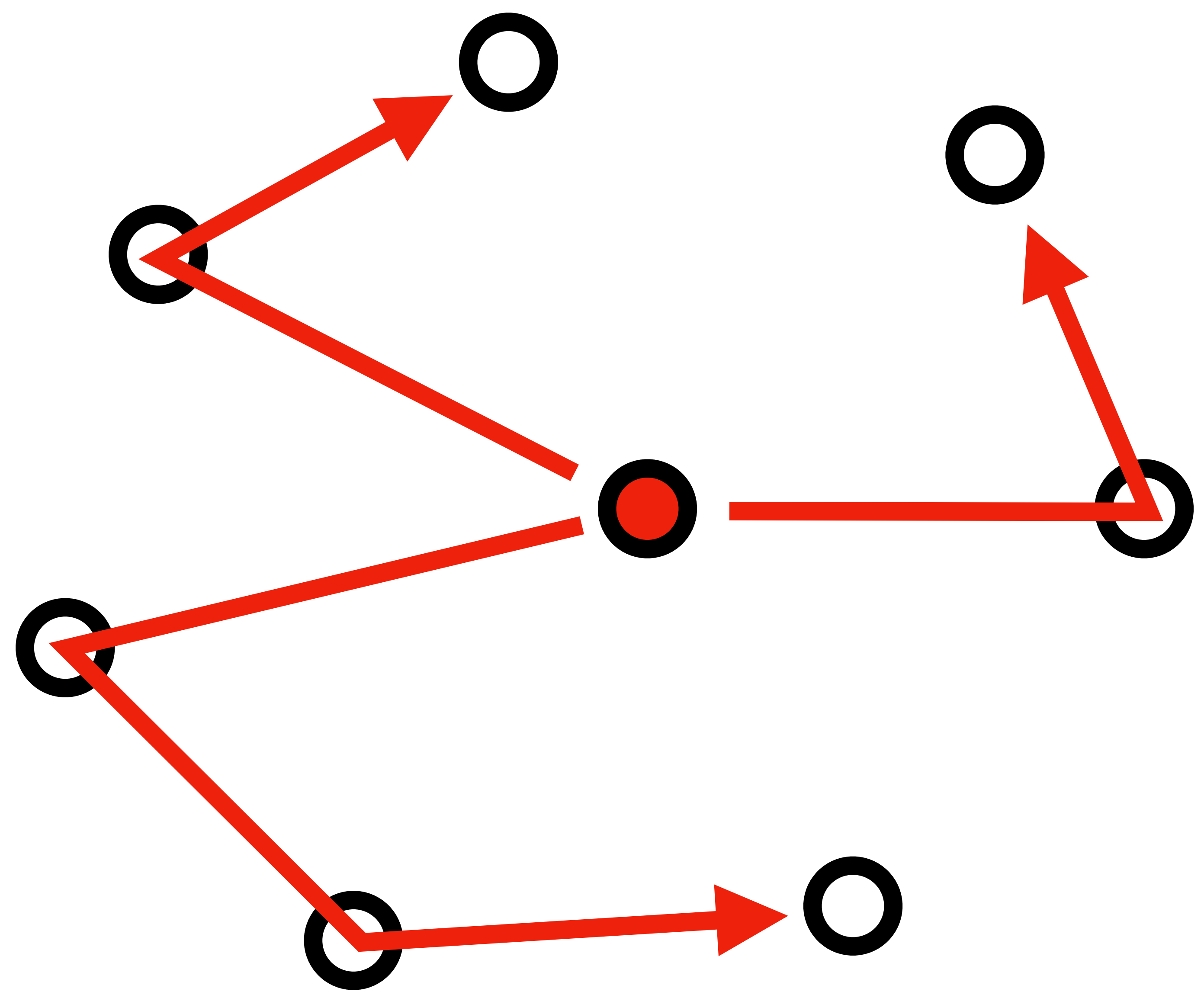
The congestion technique

Low-congestion shortest paths [Thorup'05]

1. Pick a vertex v that maximizes cg(v)
2. **Compute h-hop shortest paths at v using Bellman-Ford**
3. Add h-hop paths to Π, update cg(.)
4. Remove v from graph, go to Step 1

$\Pi = \{\pi_{s,t} \mid s, t \in V\}$ a set of short paths

cg(v) = #paths in Π containing v



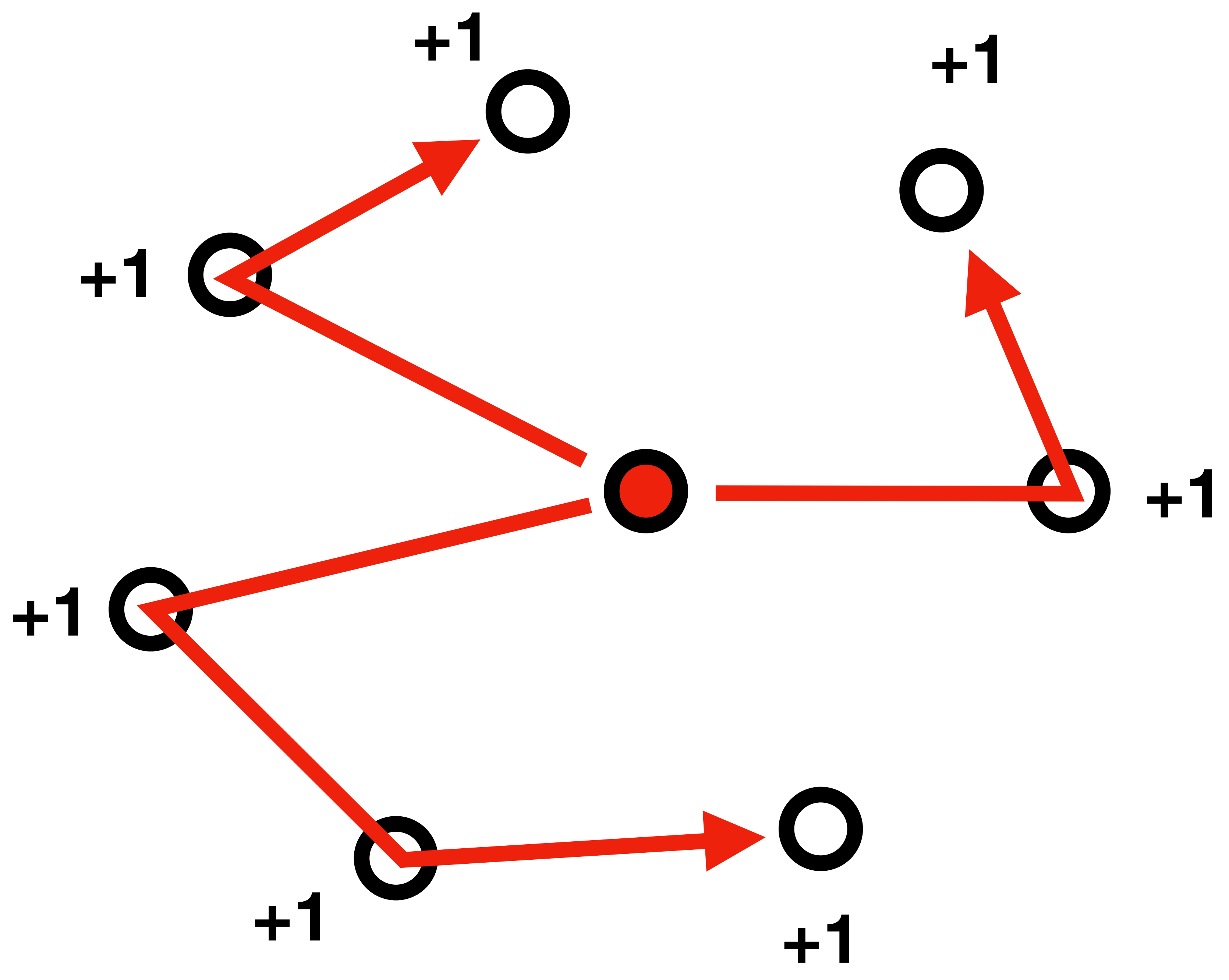
The congestion technique

Low-congestion shortest paths [Thorup'05]

1. Pick a vertex v that maximizes cg(v)
2. Compute h-hop shortest paths at v using Bellman-Ford
3. **Add h-hop paths to Π, update cg(.)**
4. Remove v from graph, go to Step 1

$\Pi = \{\pi_{s,t} \mid s, t \in V\}$ a set of short paths

cg(v) = #paths in Π containing v



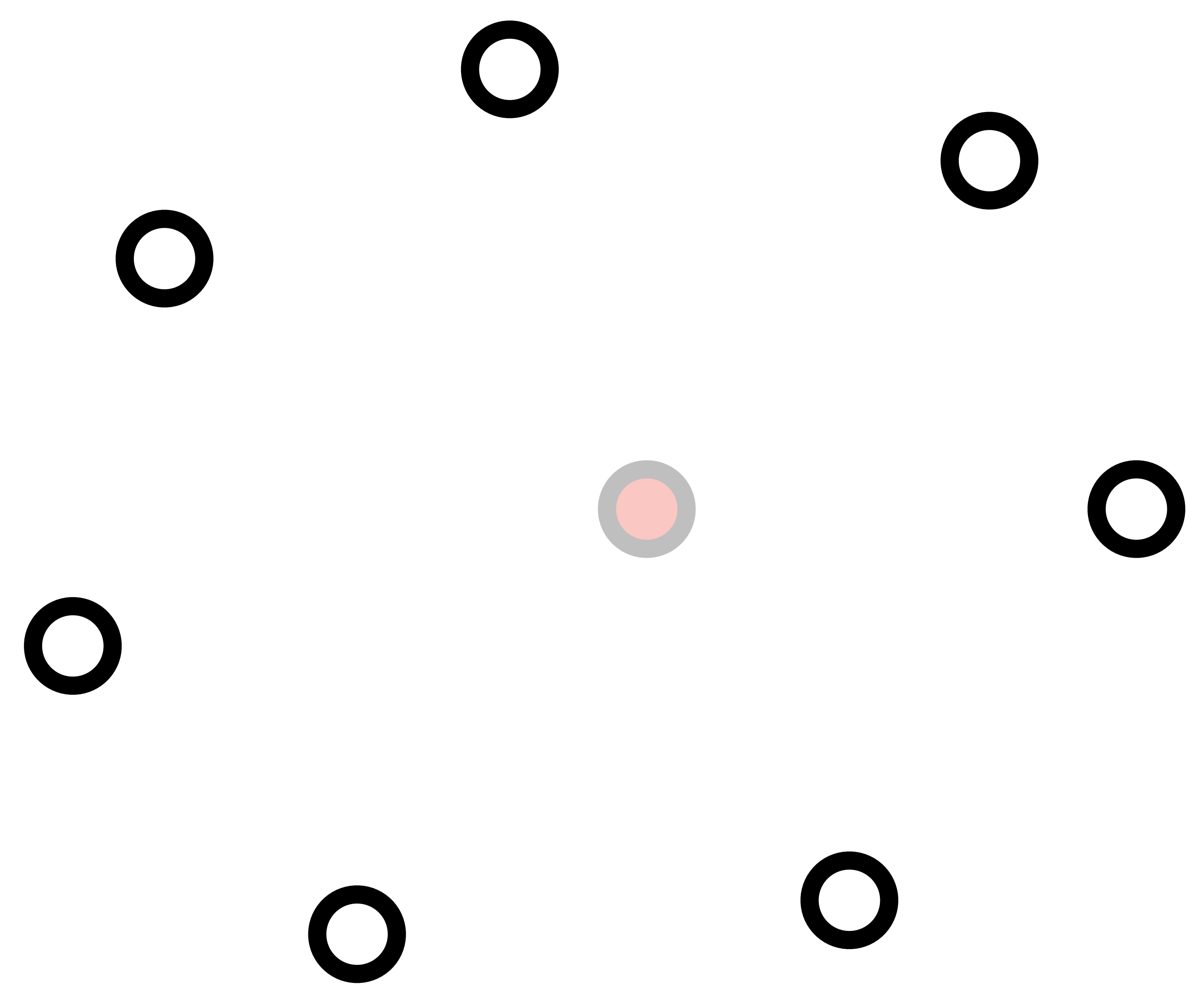
The congestion technique

Low-congestion shortest paths [Thorup'05]

1. Pick a vertex v that maximizes cg(v)
2. Compute h-hop shortest paths at v using Bellman-Ford
3. Add h-hop paths to Π, update cg(.)
4. **Remove v from graph, go to Step 1**

$\Pi = \{\pi_{s,t} \mid s, t \in V\}$ a set of short paths

cg(v) = #paths in Π containing v



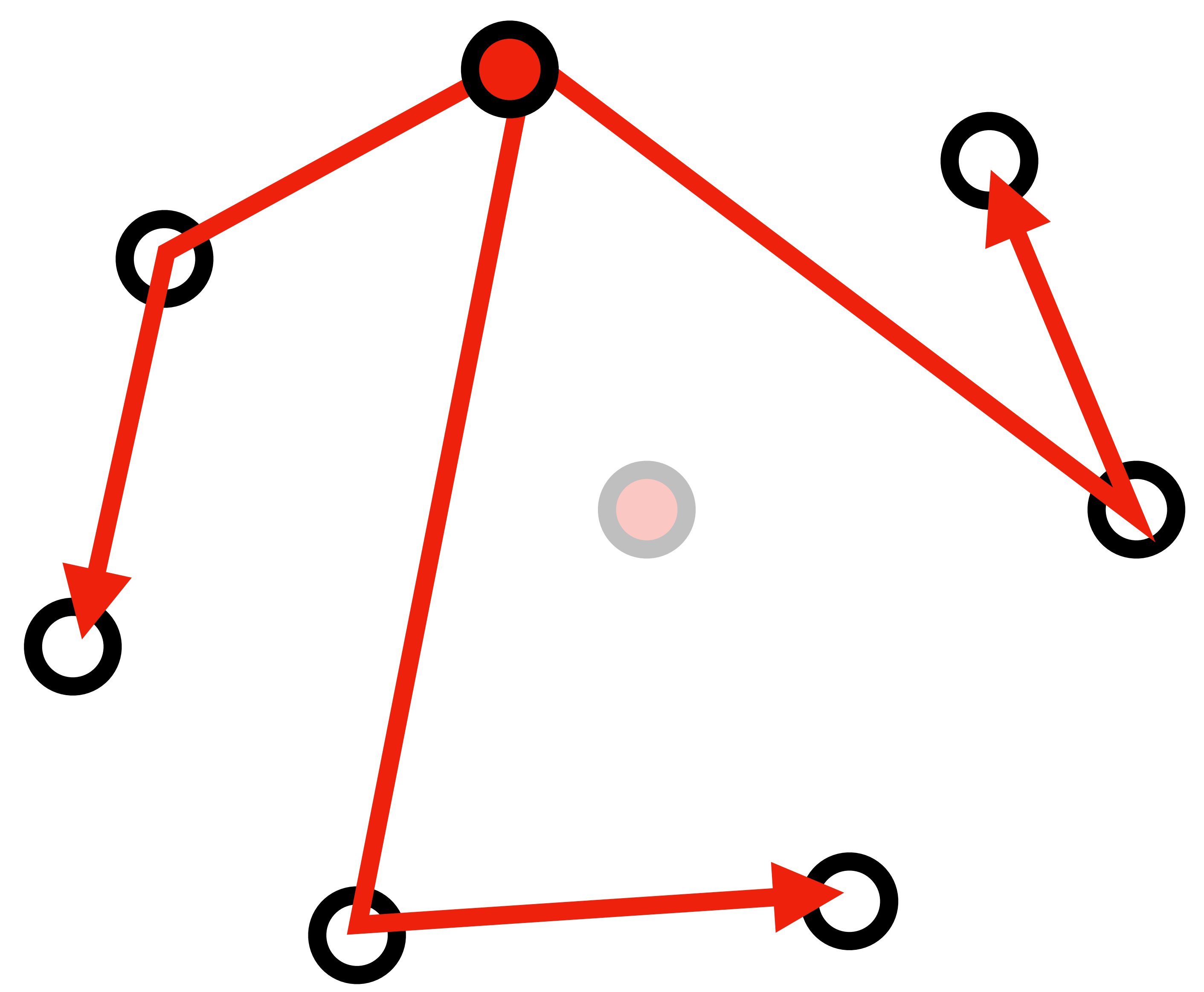
The congestion technique

Low-congestion shortest paths [Thorup'05]

1. Pick a vertex v that maximizes cg(v)
2. Compute h-hop shortest paths at v using Bellman-Ford
3. Add h-hop paths to Π, update cg(.)
4. **Remove v from graph, go to Step 1**

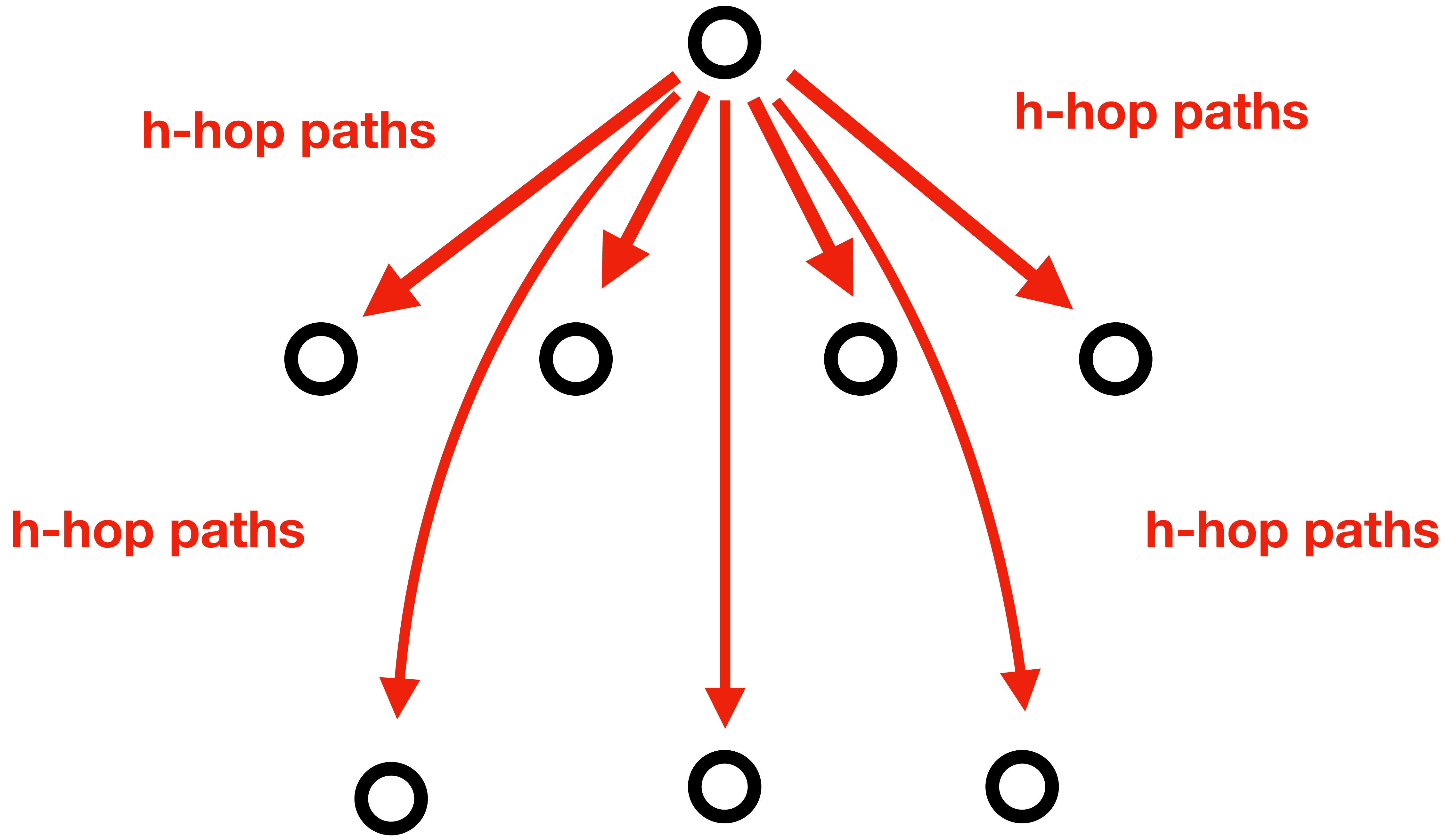
$\Pi = \{\pi_{s,t} \mid s, t \in V\}$ a set of short paths

cg(v) = #paths in Π containing v



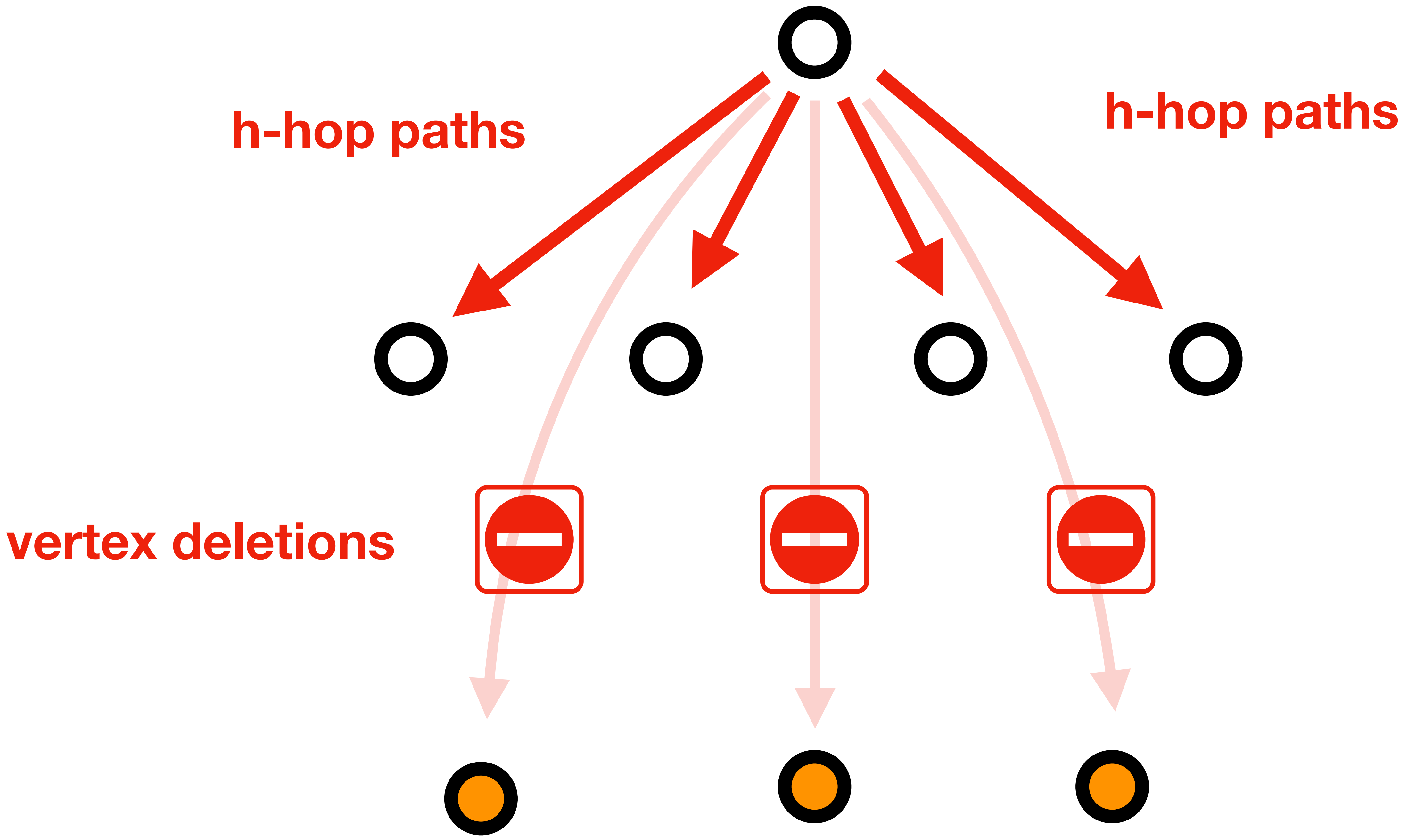
Recovery from batch deletion

Recovery by Dijkstra's algorithm [ACK'17]



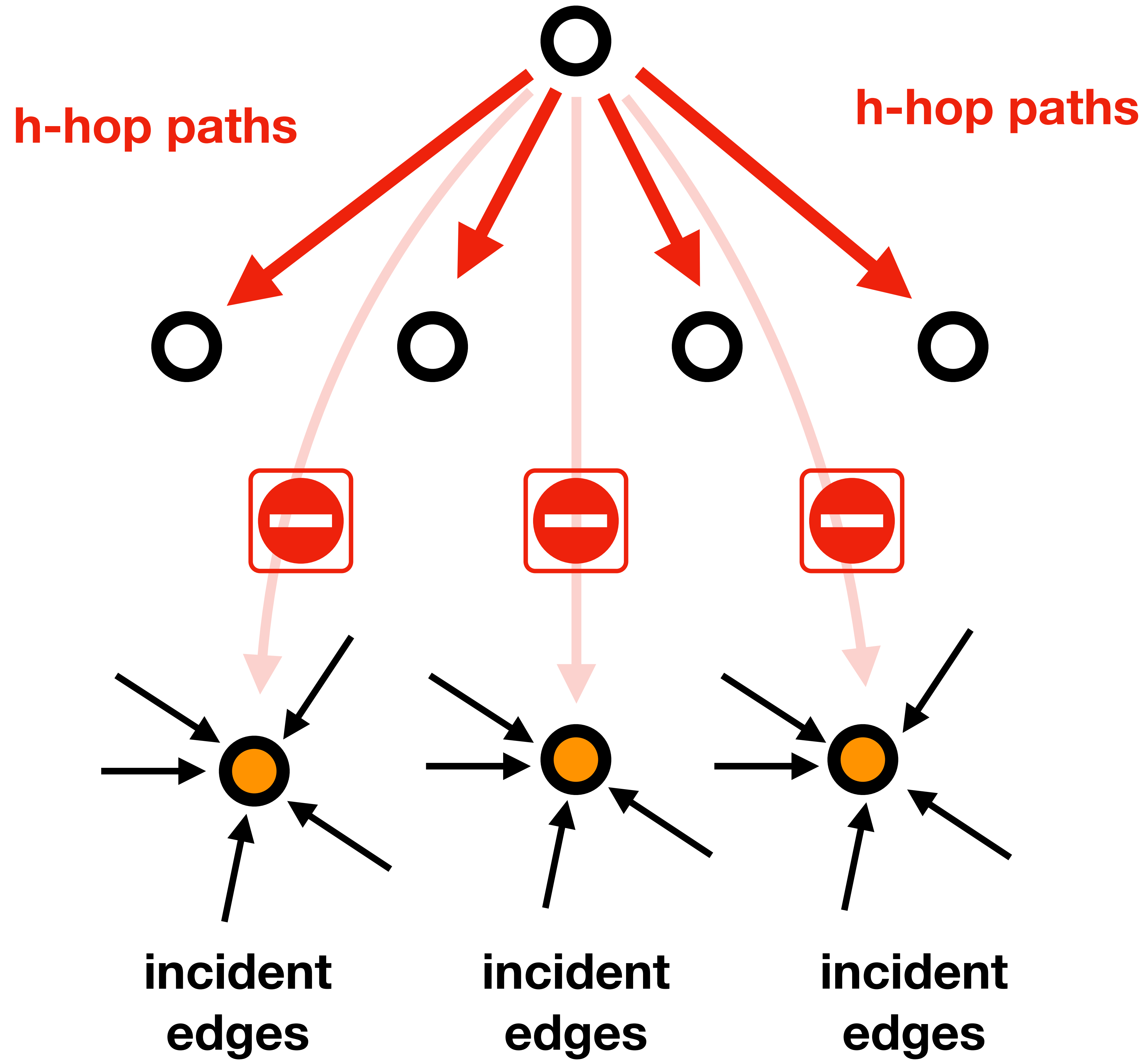
Recovery from batch deletion

Recovery by Dijkstra's algorithm [ACK'17]



Recovery from batch deletion

Recovery by Dijkstra's algorithm [ACK'17]



Recovery algorithm:

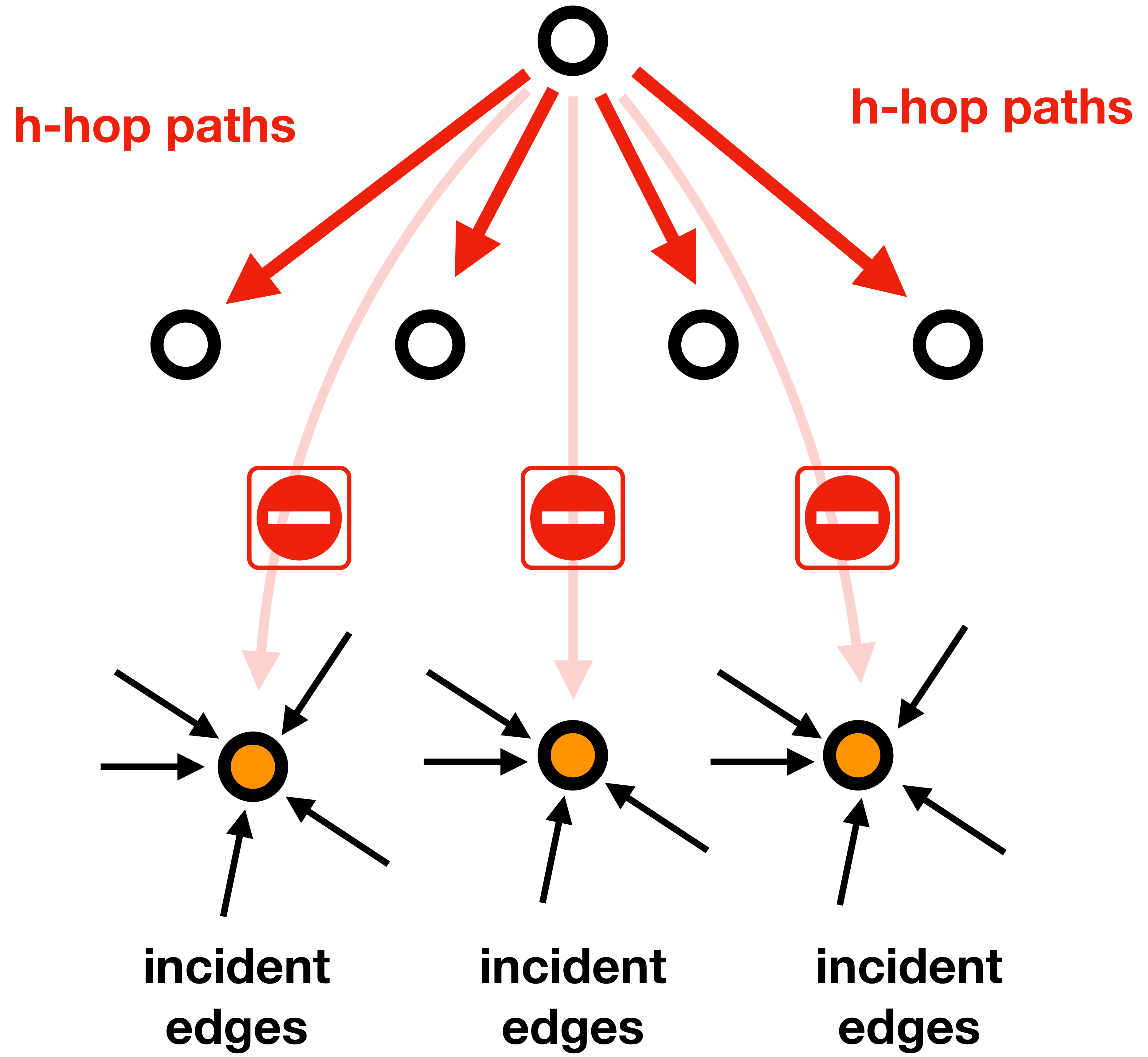
1. View red paths as **shortcuts**
2. Run Dijkstra on red / black edges

Runtime = $n \cdot$ **#destroyed** h-hop paths

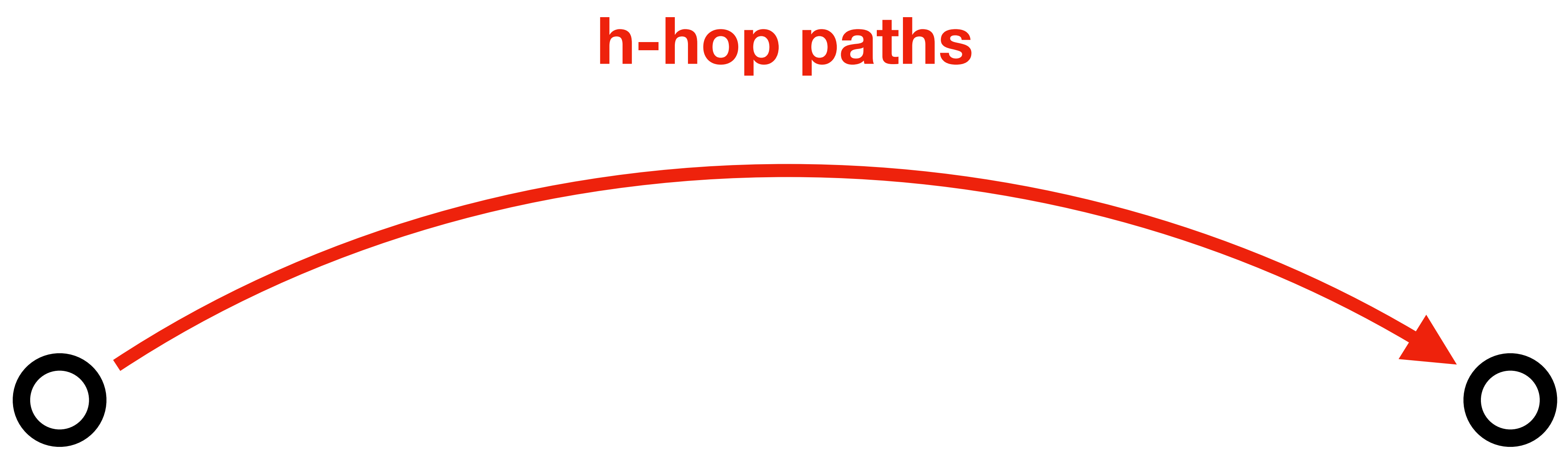
#destroyed is small by the congestion technique

Recovery from batch deletion

Recovery by Dijkstra's algorithm [ACK'17]

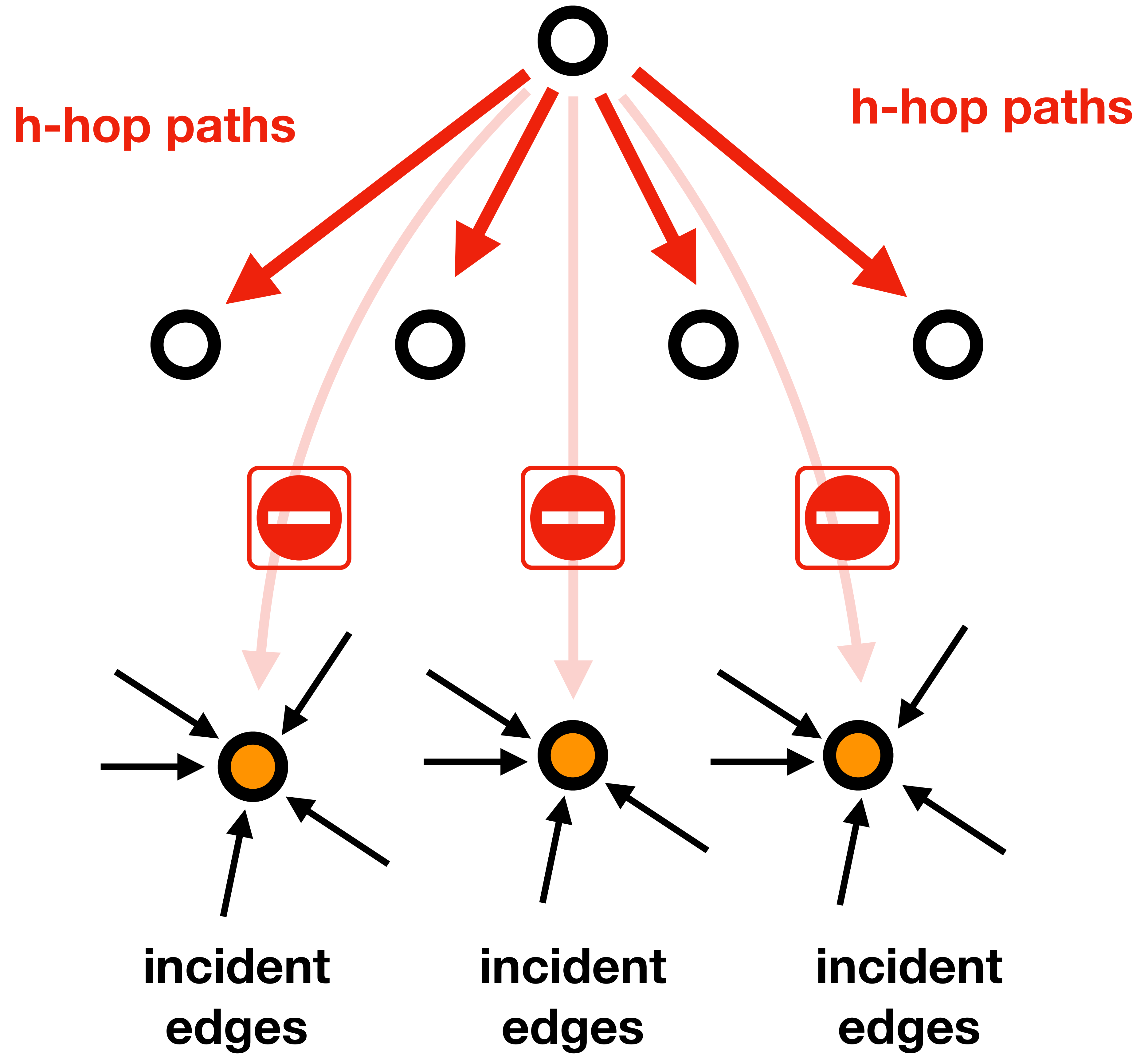


Recovery by path concat [P-WN'20]

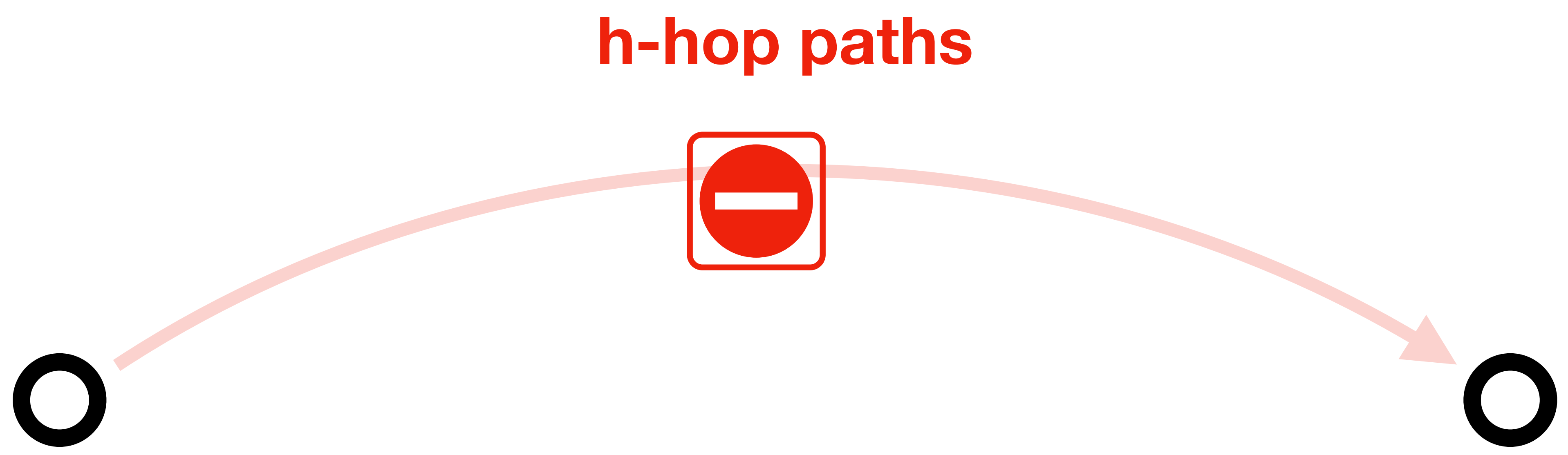


Recovery from batch deletion

Recovery by Dijkstra's algorithm [ACK'17]

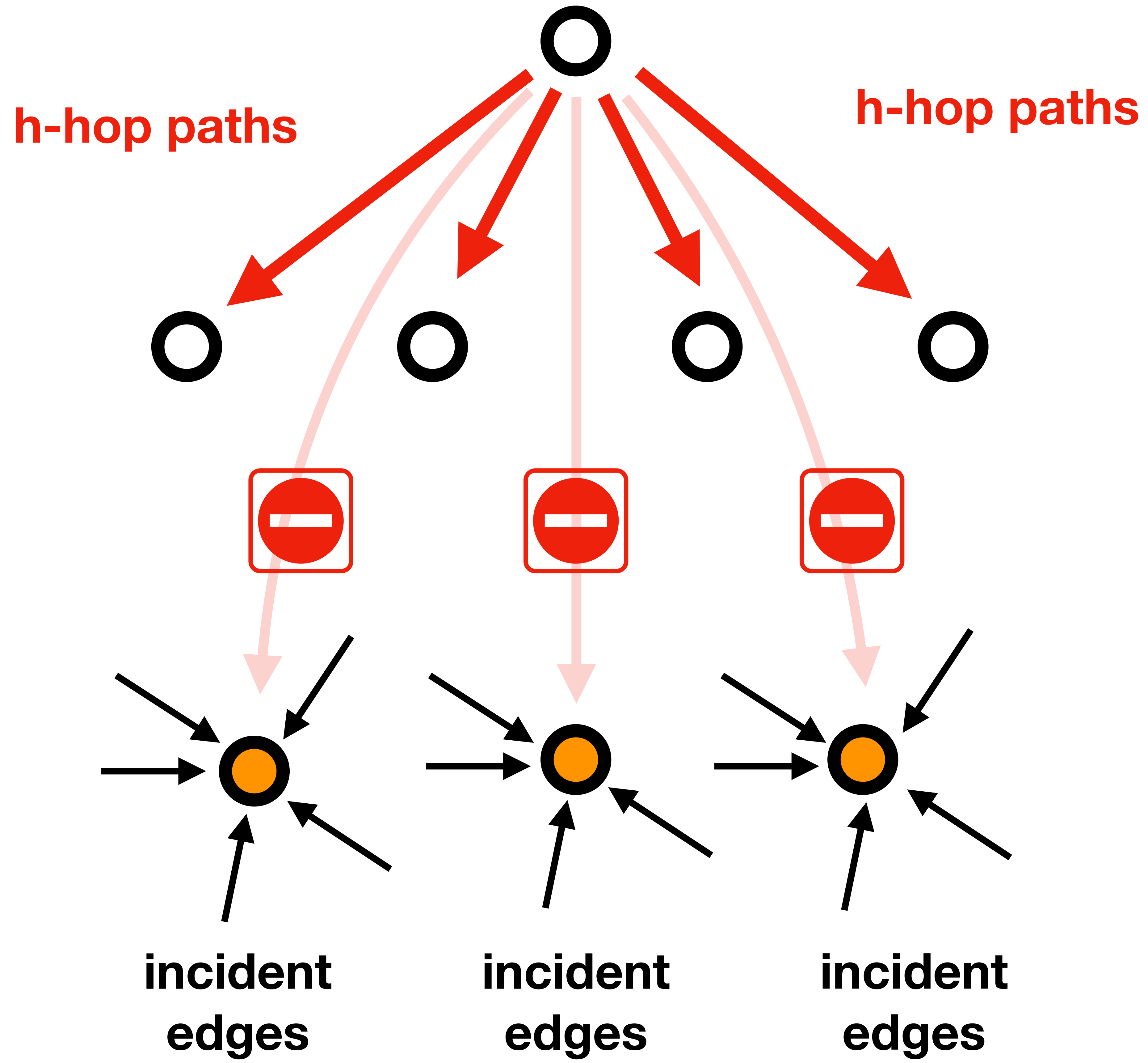


Recovery by path concat [P-WN'20]

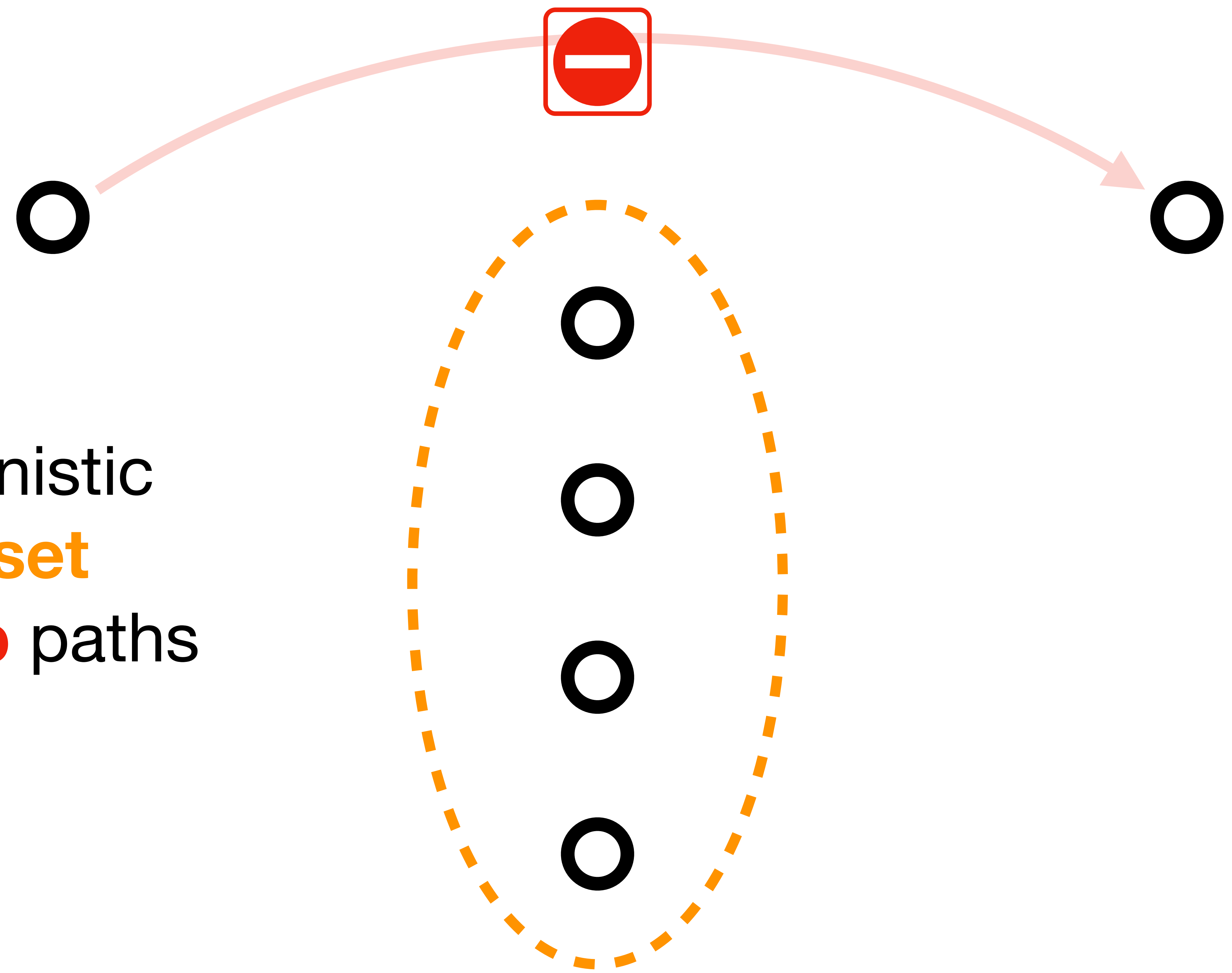


Recovery from batch deletion

Recovery by Dijkstra's algorithm [ACK'17]



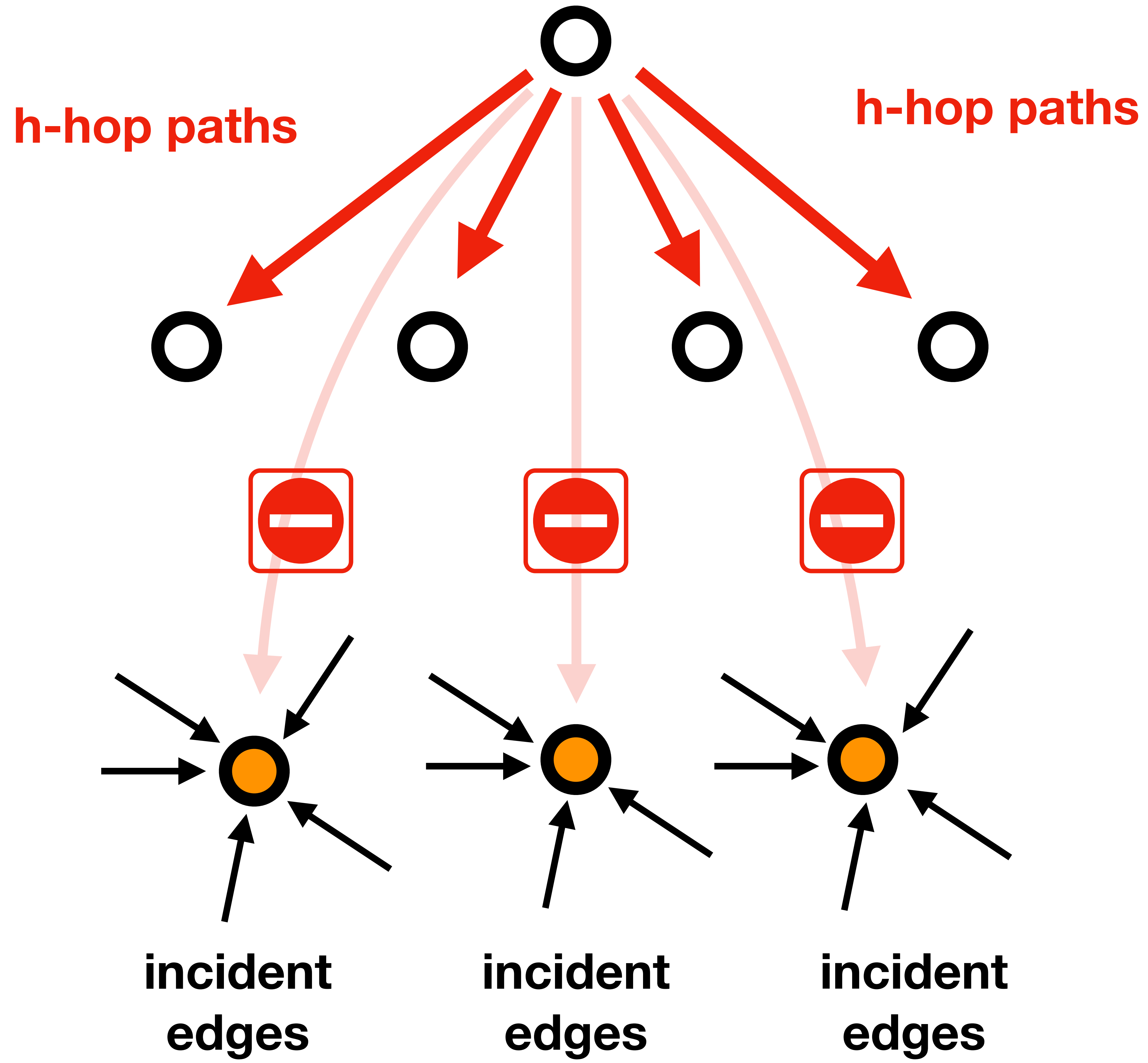
Recovery by path concat [P-WN'20]



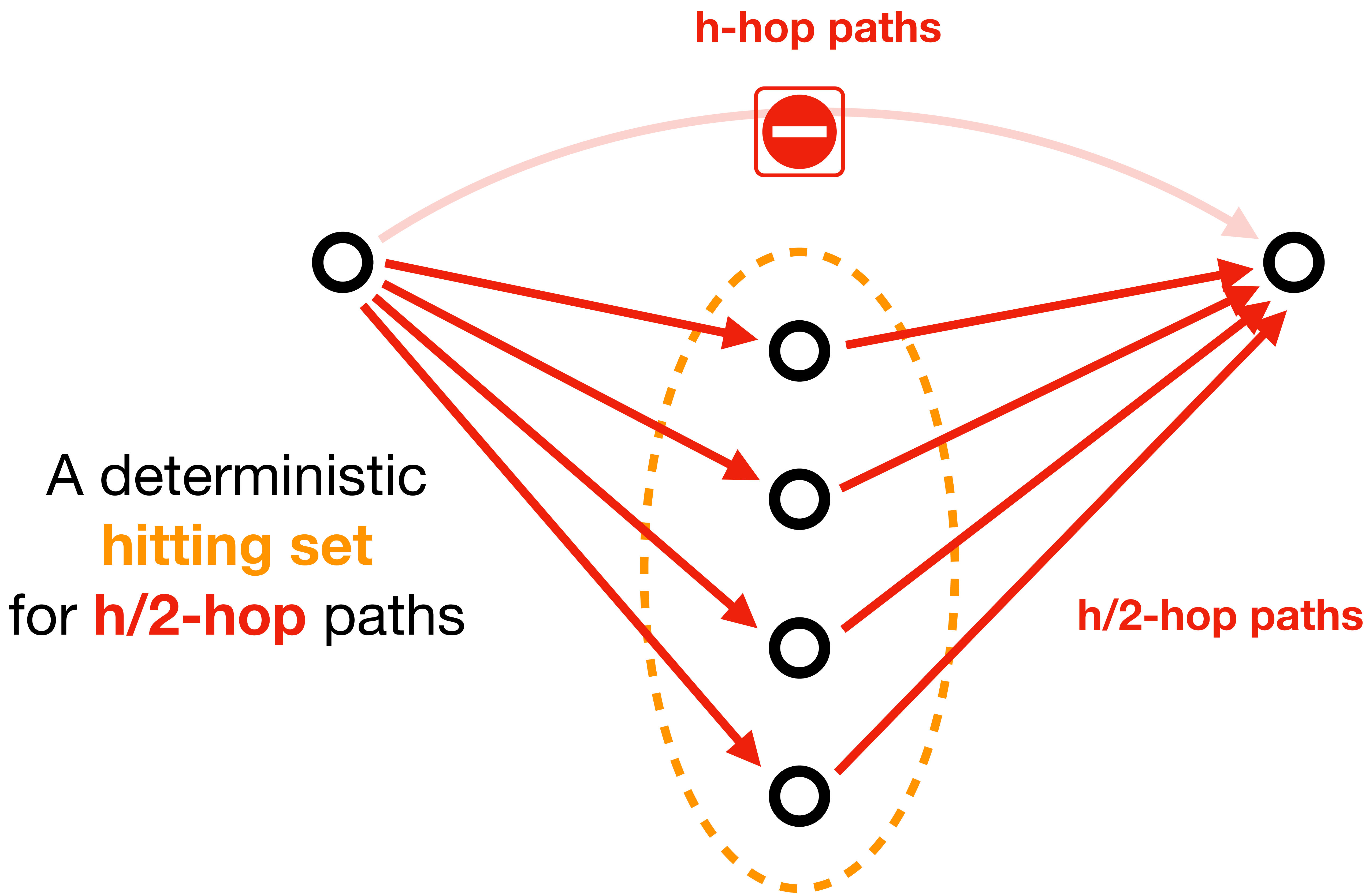
A deterministic
hitting set
for **h/2-hop** paths

Recovery from batch deletion

Recovery by Dijkstra's algorithm [ACK'17]

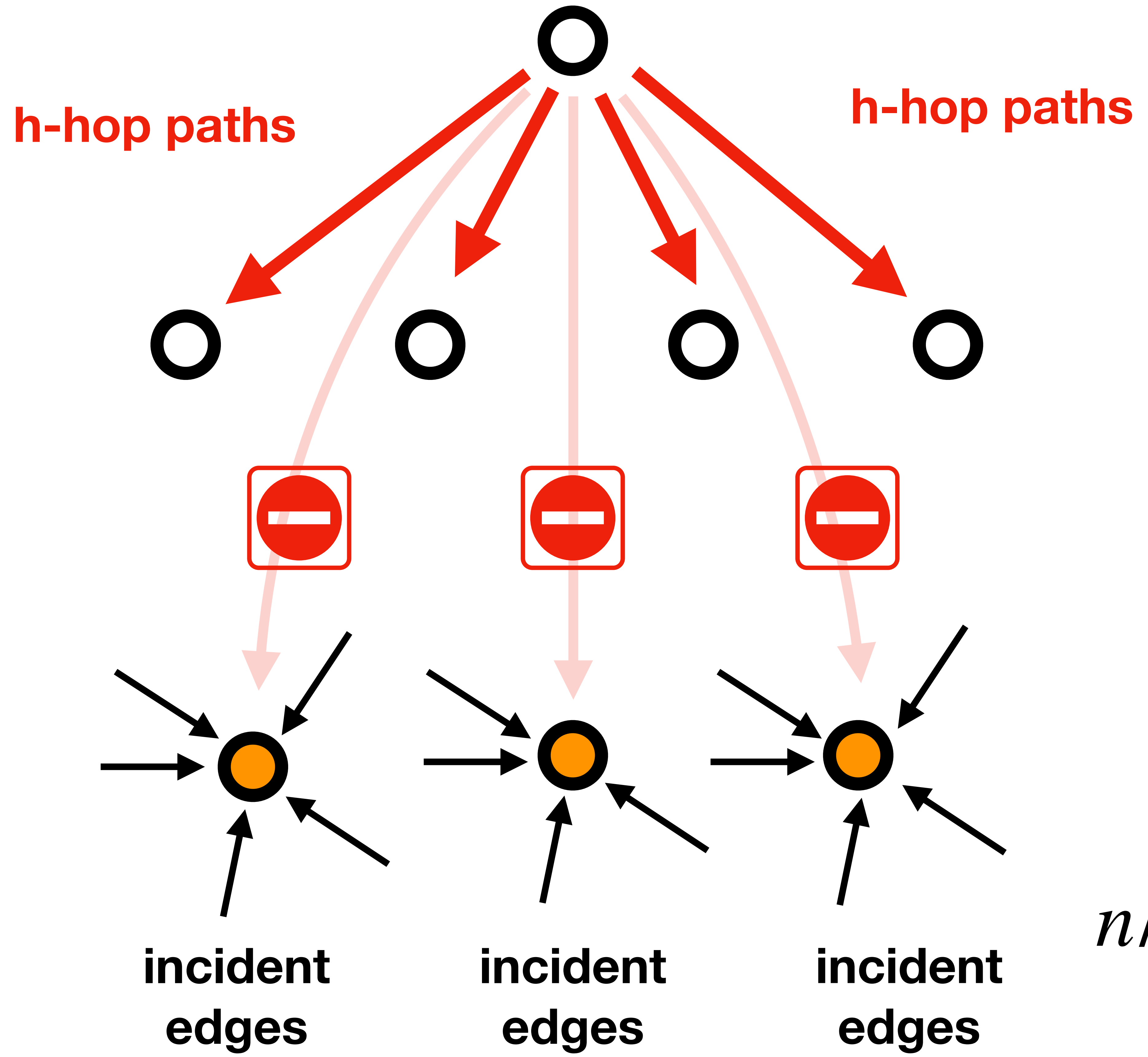


Recovery by path concat [P-WN'20]

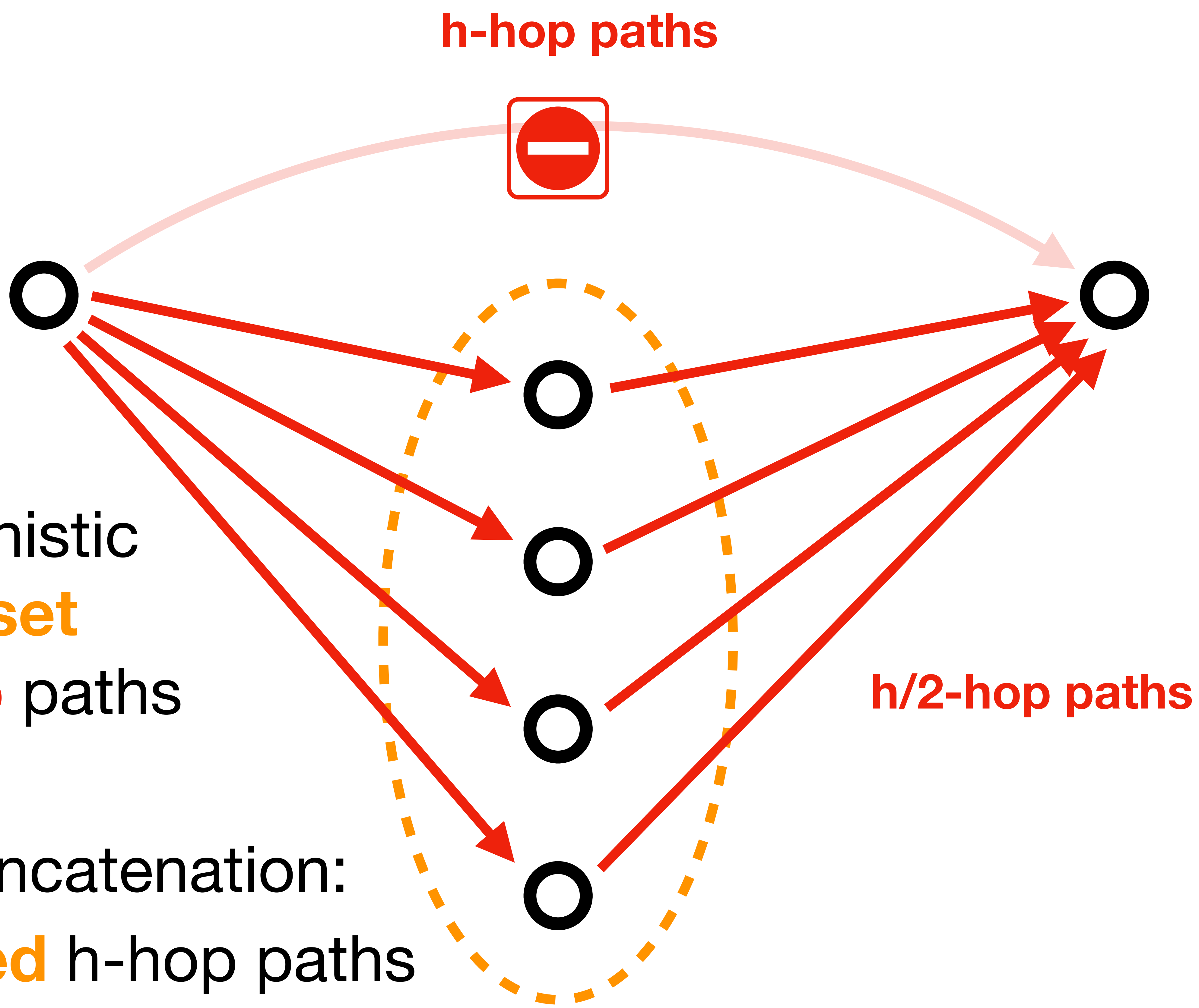


Recovery from batch deletion

Recovery by Dijkstra's algorithm [ACK'17]



Recovery by path concat [P-WN'20]

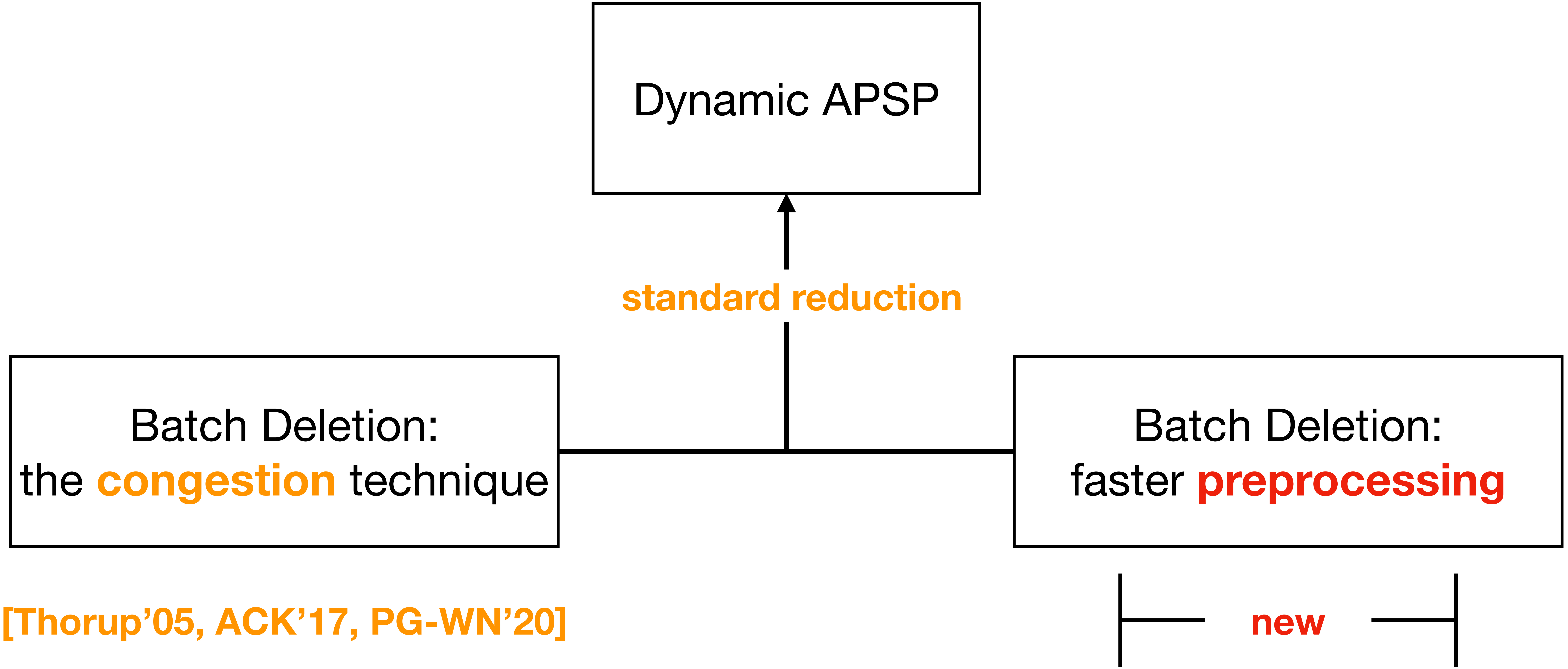


A deterministic
hitting set
for **h/2-hop** paths

Runtime of concatenation:
$n/h \cdot$ **#destroyed** h-hop paths

Our improvement

Outline


Dynamic APSP
standard reduction
Batch Deletion:
the congestion technique
[Thorup’05, ACK’17, PG-WN’20]
Batch Deletion:
faster preprocessing
new

Decremental hop-restricted shortest paths

Low-congestion shortest paths [Thorup'05]

1. Pick a vertex v that maximizes cg(v)
2. Compute h-hop shortest paths at v using Bellman-Ford
3. Add h-hop paths to Π, update cg(.)
4. Remove v from graph, go to Step 1

Decremental hop-restricted shortest paths

Low-congestion shortest paths [Thorup'05]

1. Pick a vertex v that maximizes cg(v)

2. **Compute h-hop shortest paths at v using Bellman-Ford**

3. Add h-hop paths to Π, update cg(.)

4. **Remove v from graph, go to Step 1**

Decremental hop-restricted shortest paths

Low-congestion shortest paths [Thorup'05]

1. Pick a vertex v that maximizes cg(v)
2. **Compute h-hop shortest paths at v using Bellman-Ford**
3. Add h-hop paths to Π, update cg(.)
4. **Remove v from graph, go to Step 1**

Decremental h-hop shortest paths:

1. Adversary picks a vertex v
2. Compute **h-hop SSSP at v**
3. Adversary **deletes an arbitrary** vertex
4. Go to Step 1

Decremental hop-restricted shortest paths

Trivial algorithm:

- Apply Bellman-Ford for h-hop SSSP
- Total time = $n^2 h \cdot$ **#deletions**

Faster runtime?

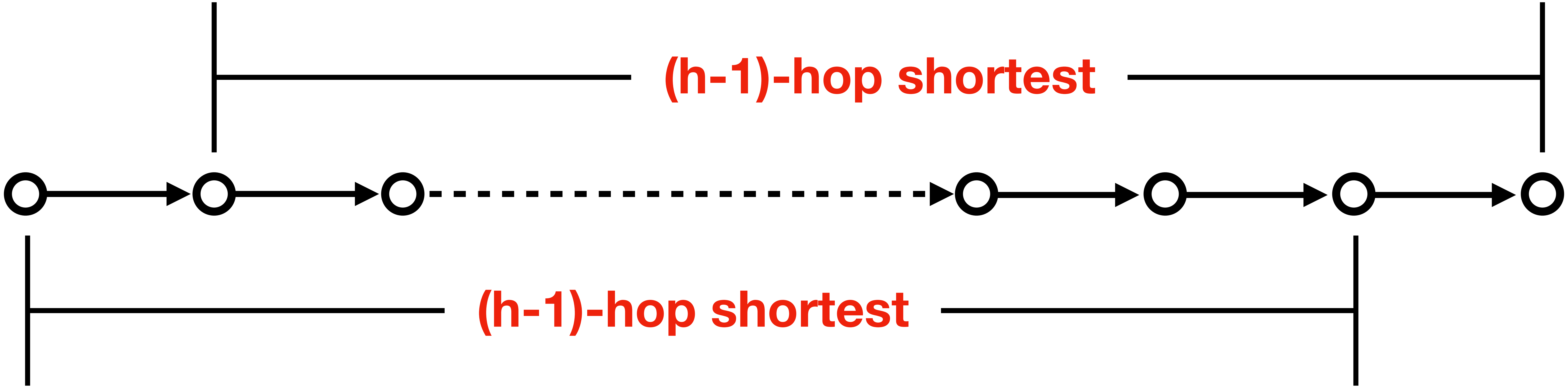
- Try to **maintain all h-hop paths** under vertex deletions

Decremental h-hop shortest paths:

1. Adversary picks a vertex v
2. Compute **h-hop SSSP at v**
3. Adversary **deletes an arbitrary** vertex
4. Go to Step 1

Locally h-hop shortest paths

- Adapt the idea of **locally shortest paths** in [Demetrescu and Italiano, 2004]
- A path $\langle u_0, u_1, \cdots, u_k \rangle$ is **locally h-hop shortest**, if both of the sub-paths $\langle u_0, u_1, \cdots, u_{k-1} \rangle$ and $\langle u_1, \cdots, u_k \rangle$ are **(h-1)-hop shortest paths**

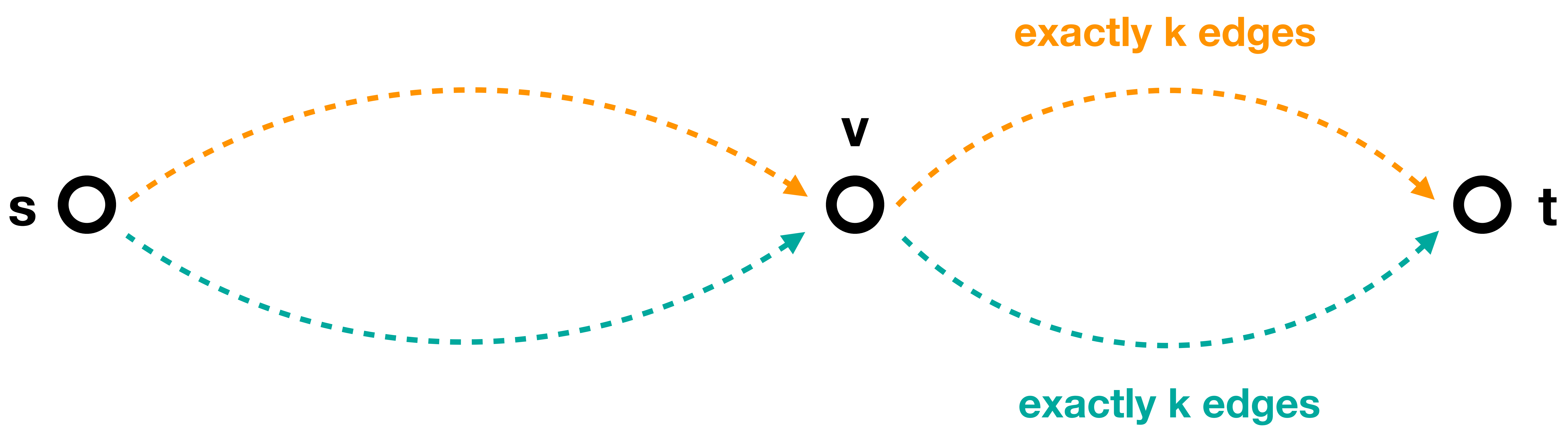


Locally h-hop shortest paths

Shortest locally h-hop shortest paths = h-hop shortest paths

#(locally h-hop) can be bounded

- Each vertex v is on **at most h** different **locally h-hop paths** from s to t
- At most $n^3 \log n$ **(all-pairs locally h-hop) in total**

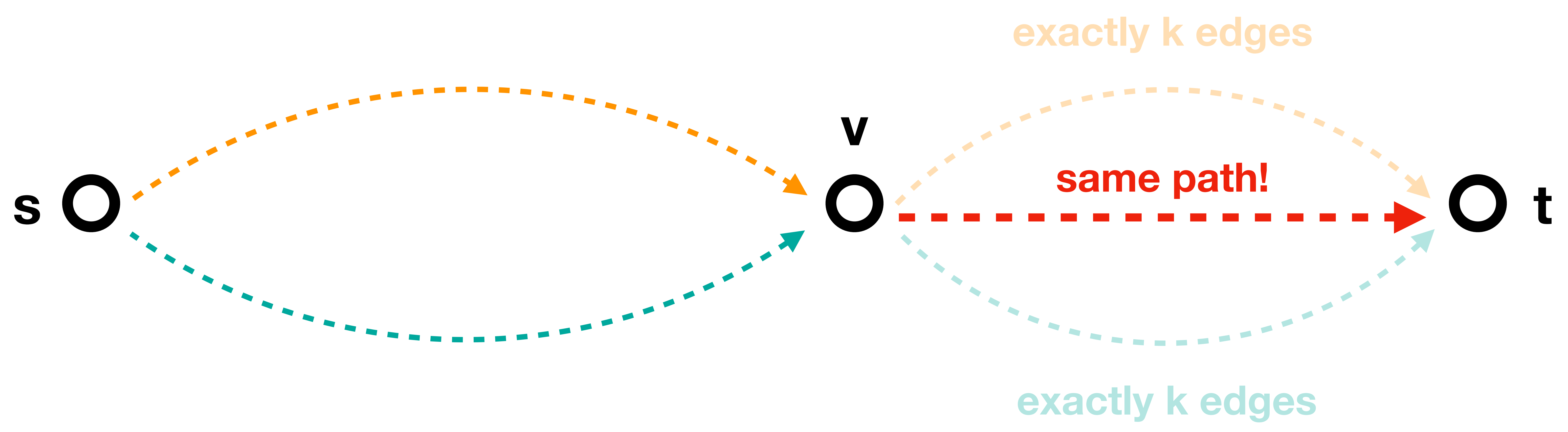


Locally h-hop shortest paths

Shortest locally h-hop shortest paths = h-hop shortest paths

#(locally h-hop) can be bounded

- Each vertex v is on **at most h** different **locally h-hop paths** from s to t
- At most $n^3 \log n$ **(all-pairs locally h-hop) in total**

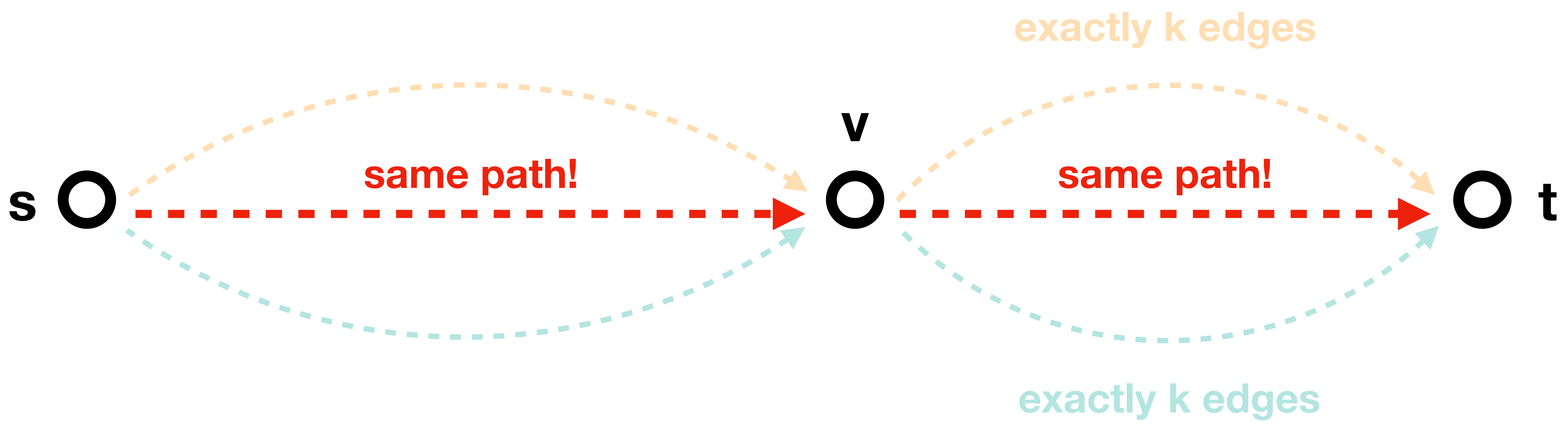


Locally h-hop shortest paths

Shortest locally h-hop shortest paths = h-hop shortest paths

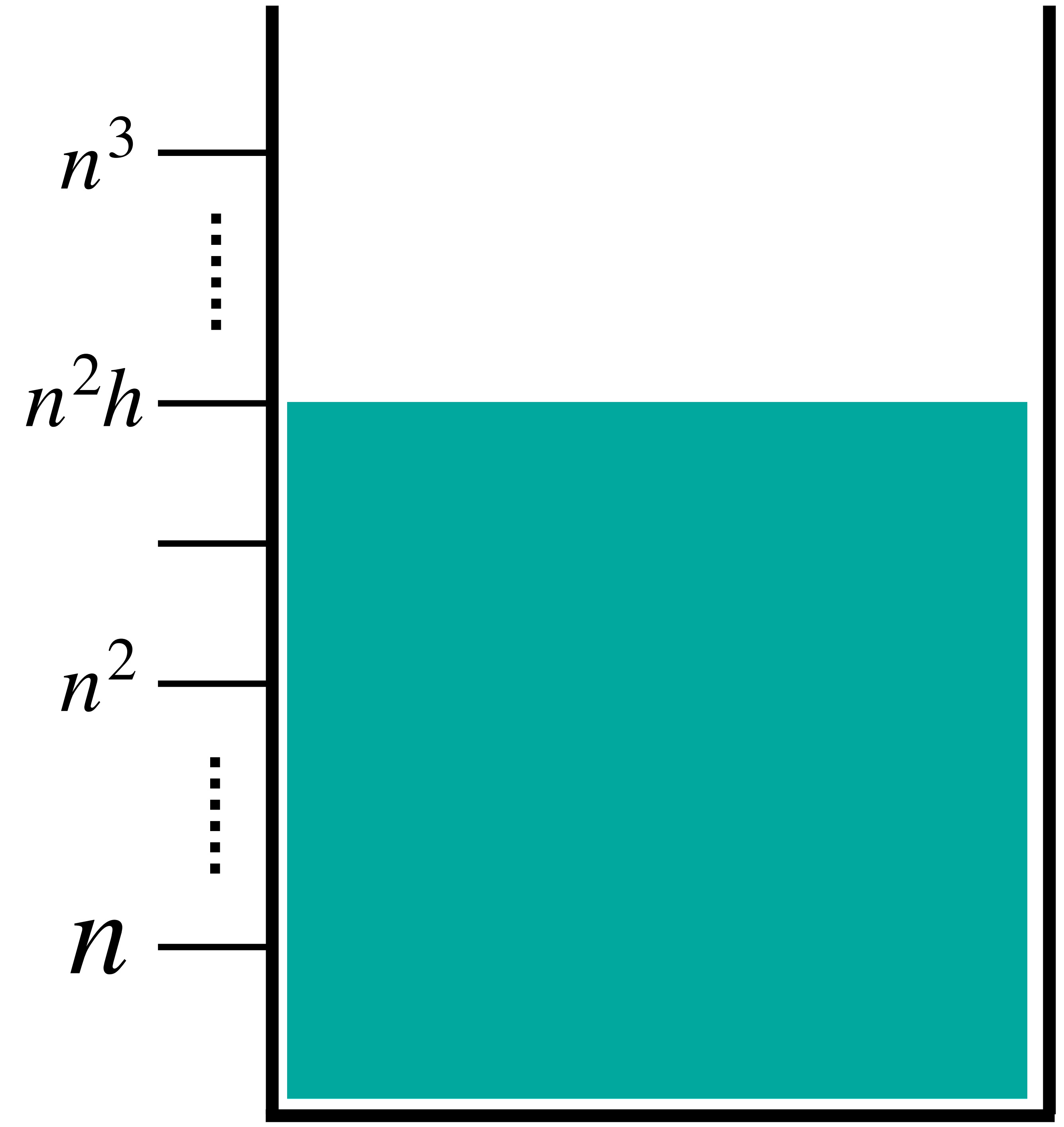
#(locally h-hop) can be bounded

- Each vertex v is on **at most h** different **locally h-hop paths** from s to t
- At most $n^3 \log n$ **(all-pairs locally h-hop) in total**



Decremental locally h-hop shortest paths

Maintain all locally h-hop shortest paths
under **vertex deletions**



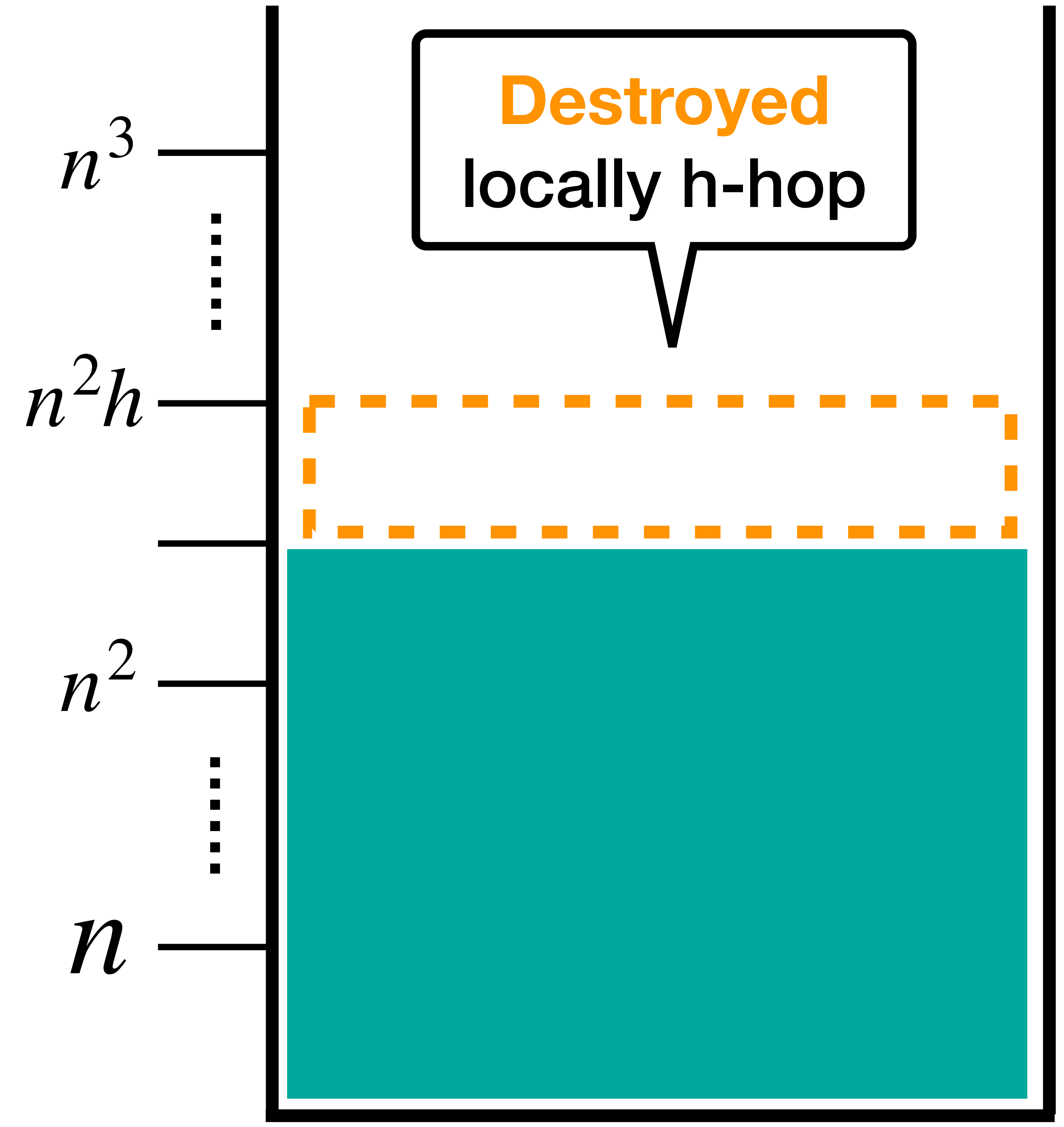
total #(locally h-hop paths)

Decremental locally h-hop shortest paths

Maintain all locally h-hop shortest paths under **vertex deletions**

1. A vertex deletion **hits** at most n^2h **old locally h-hop shortest paths**

   Runtime = #(destroyed) $\leq n^2h$



total #(locally h-hop paths)

Decremental locally h-hop shortest paths

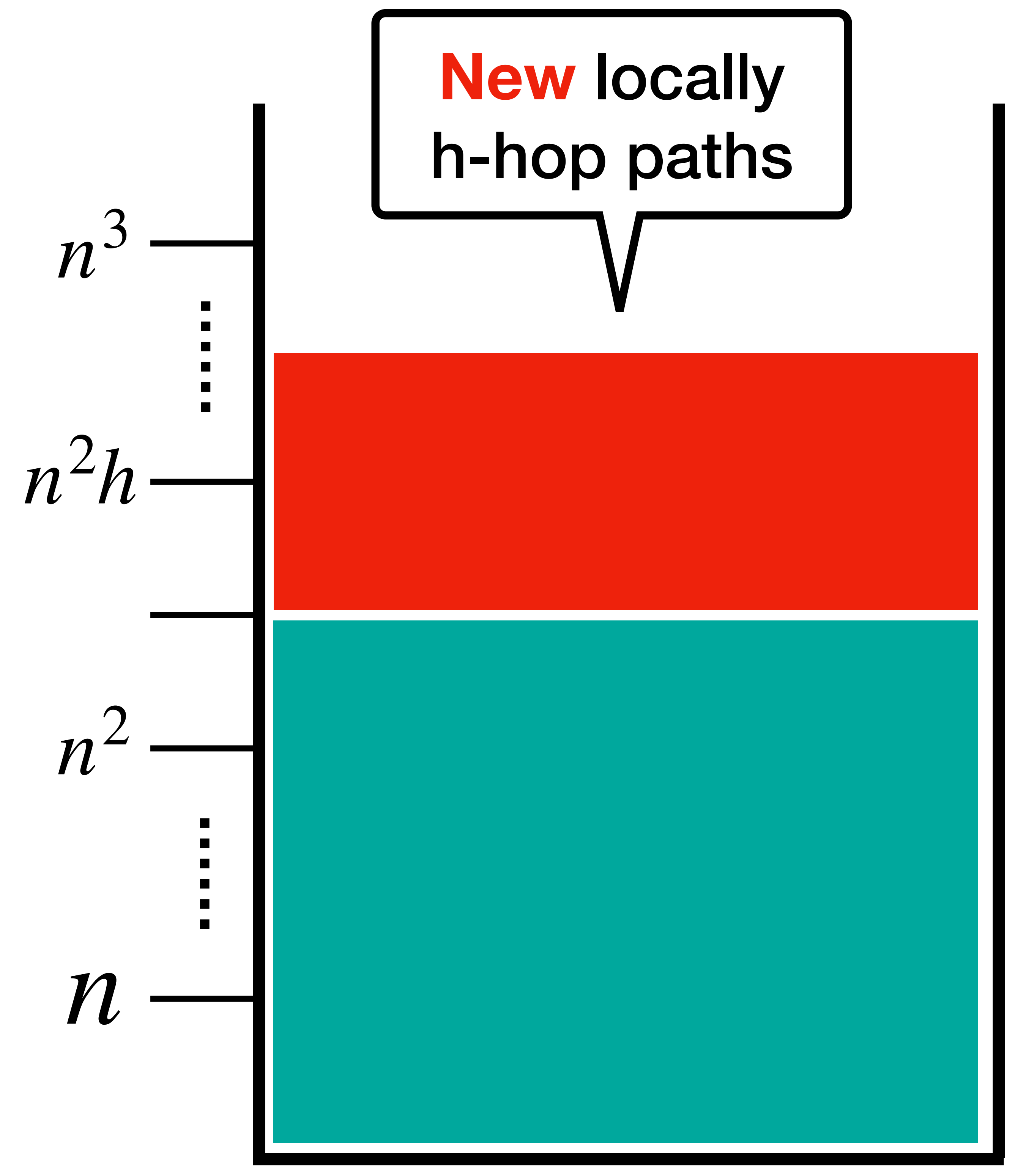
Maintain all locally h-hop shortest paths under **vertex deletions**

1. A vertex deletion **hits** at most n^2h **old locally h-hop shortest paths**

   Runtime = #(destroyed) $\leq n^2h$

2. A vertex deletion may generate **new locally h-hop shortest paths**

   Runtime = #(new locally h-hop)



total #(locally h-hop paths)

Decremental locally h-hop shortest paths

Maintain all locally h-hop shortest paths under **vertex deletions**

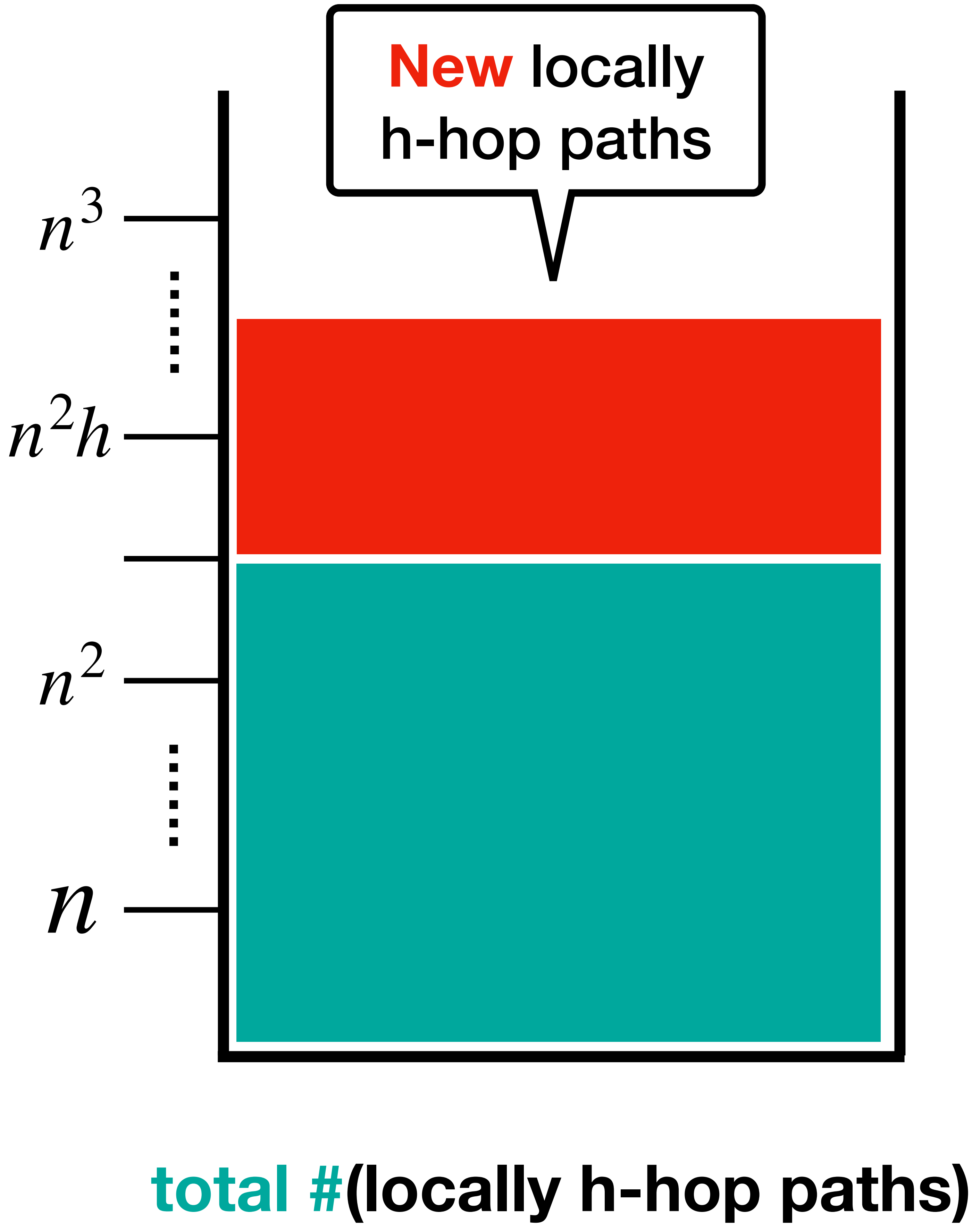
1. A vertex deletion **hits** at most n^2h old locally h-hop shortest paths

   Runtime = #(destroyed) $\leq n^2h$

2. A vertex deletion may generate new locally h-hop shortest paths

   Runtime = #(new locally h-hop)

3. Output-sensitive —> total time = n^3h



Accelerating Bellman-Ford

Original goal:

1. Adversary picks a vertex v
2. Compute **h-hop SSSP at v**
3. Adversary **deletes an arbitrary** vertex
4. Go to Step 1

Trivial algorithm:

- Apply Bellman-Ford for h-hop SSSP
- Total time = $n^2 h \cdot$ **#deletions**

Faster runtime?

- Decremental h-hop paths has total runtime = $n^3 h$, **no improvement**

Accelerating Bellman-Ford

Original goal:

1. Adversary picks a vertex v
2. Compute **h-hop SSSP at v**
3. Adversary **deletes an arbitrary** vertex
4. Go to Step 1

Faster runtime?

- Decremental h-hop paths has total runtime = n^3h, **no improvement**

Accelerating Bellman-Ford

Original goal:

1. Adversary picks a vertex v
2. Compute **h-hop SSSP at v**
3. Adversary **deletes an arbitrary** vertex
4. Go to Step 1

Faster runtime?

- Decremental h-hop paths has total runtime = $n^3 h$, **no improvement**

Solution:

- Apply decremental g-hop paths
- Bellman-Ford runs in time $n^2 h / g$
- Total time = $n^3(g + h/g) < n^3 h$

Accelerating Bellman-Ford

Original goal:

1. Adversary picks a vertex v
2. Compute **h-hop SSSP at v**
3. Adversary **deletes an arbitrary** vertex
4. Go to Step 1

Faster runtime?

- Decremental h-hop paths has total runtime = n^3h, **no improvement**

Solution:

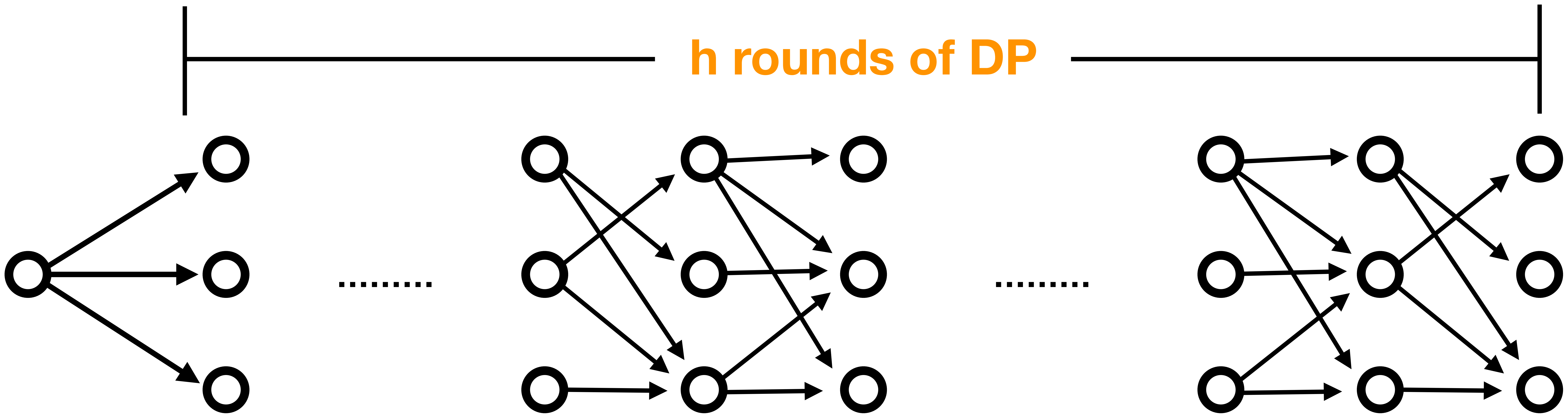
- Apply decremental g-hop paths
- Bellman-Ford runs in time n^2h/g
- Total time = $n^3(g + h/g) < n^3h$

Accelerating Bellman-Ford

Bellman-Ford runs in time n^2h/g

1. Standard Bellman-Ford = **h rounds** of dynamic programming

2. **Compress every g-rounds** into a single round using g-hop paths

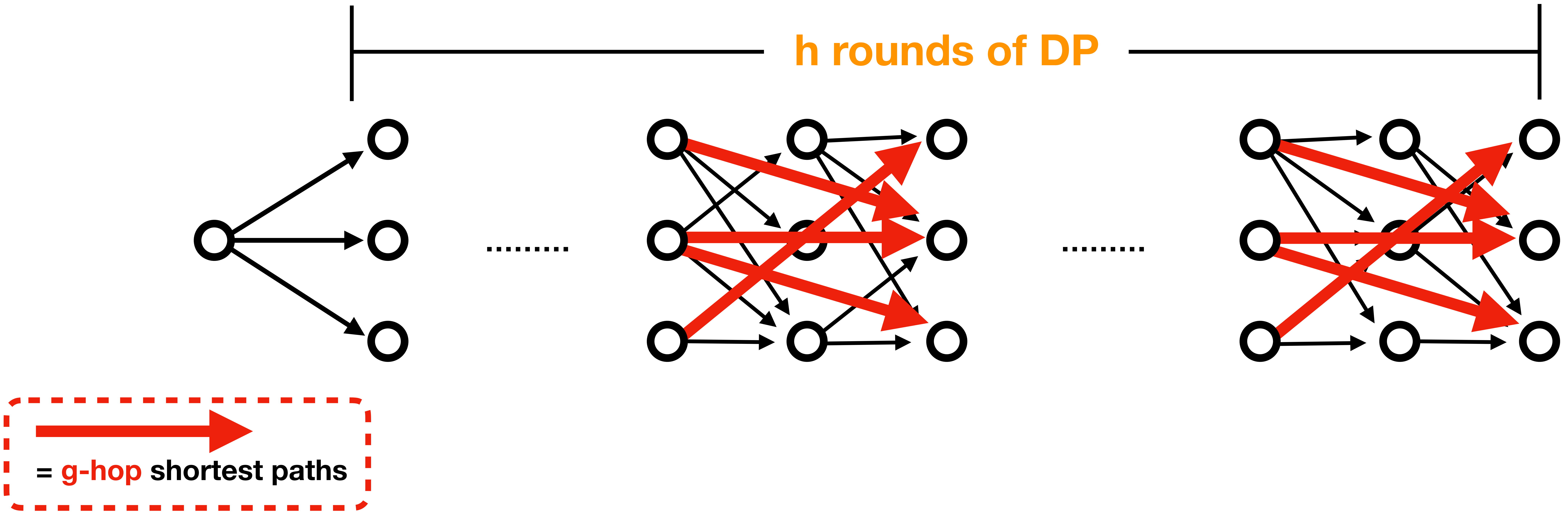


Accelerating Bellman-Ford

Bellman-Ford runs in time n^2h/g

1. Standard Bellman-Ford = **h rounds** of dynamic programming

2. **Compress every g-rounds** into a single round using g-hop paths

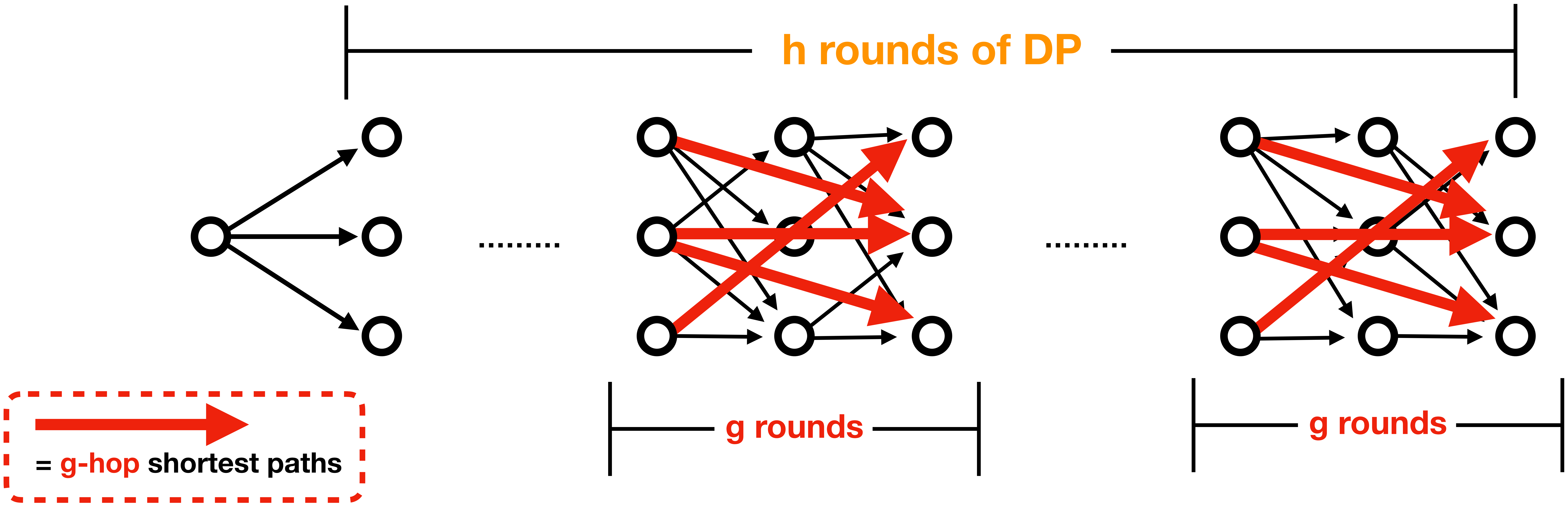


Accelerating Bellman-Ford

Bellman-Ford runs in time n^2h/g

1. Standard Bellman-Ford = **h rounds** of dynamic programming

2. **Compress every g-rounds** into a single round using g-hop paths



Further Questions

- Faster randomized worst-case update time $n^{3-1/3-\epsilon}$?
- Faster deterministic worst-case update time $n^{3-1/3}$?

Thank you !