

Counting minimum  $(s,t)$ -cuts  
in weighted planar graphs  
in polynomial time

Ivona Bezáková

(Rochester Institute of Technology)

Adam J. Friedlander

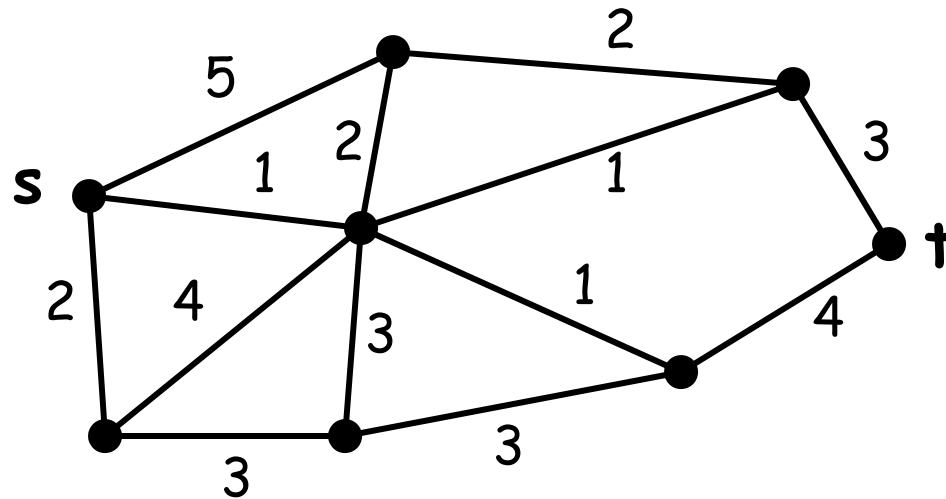
(before: B.S./M.S. RIT, now: IBM, Poughkeepsie, NY)

# The problem: Counting minimum $(s,t)$ -cuts

Input: a positively weighted (directed) planar graph  $G=(V,E)$   
and two vertices  $s,t$

Output: the number of minimum  $(s,t)$ -cuts of  $G$

Example:



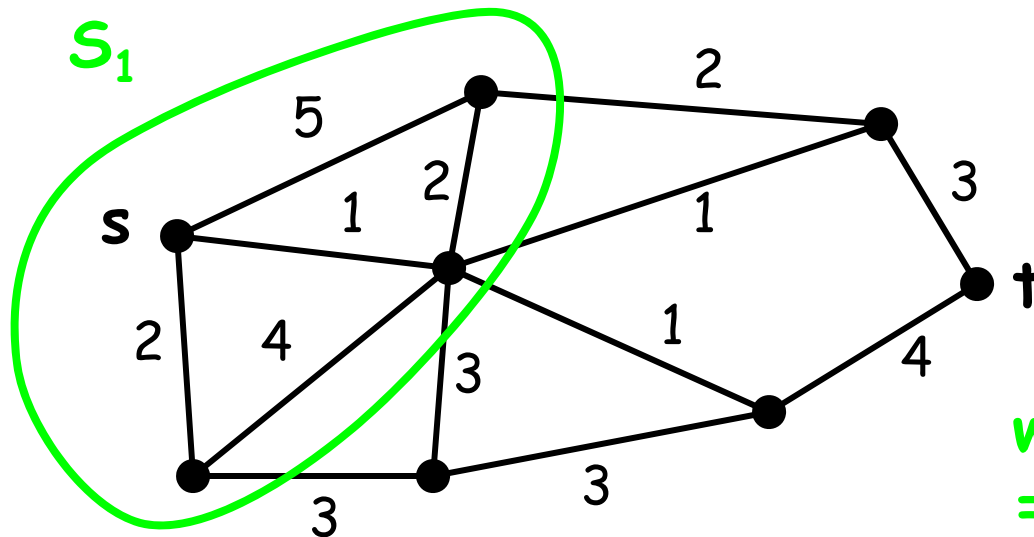
Recall:  $(s,t)$ -cut is a set  $S \subseteq V$  containing  $s$  but not  $t$ ; its weight is the sum of edge weights out of  $S$

# The problem: Counting minimum (s,t)-cuts

Input: a positively weighted (directed) planar graph  $G=(V,E)$  and two vertices  $s,t$

Output: the number of minimum (s,t)-cuts of  $G$

Example:



$$\begin{aligned} \text{weight}(S_1) &= 2+1+1+3+3 \\ &= 10 \end{aligned}$$

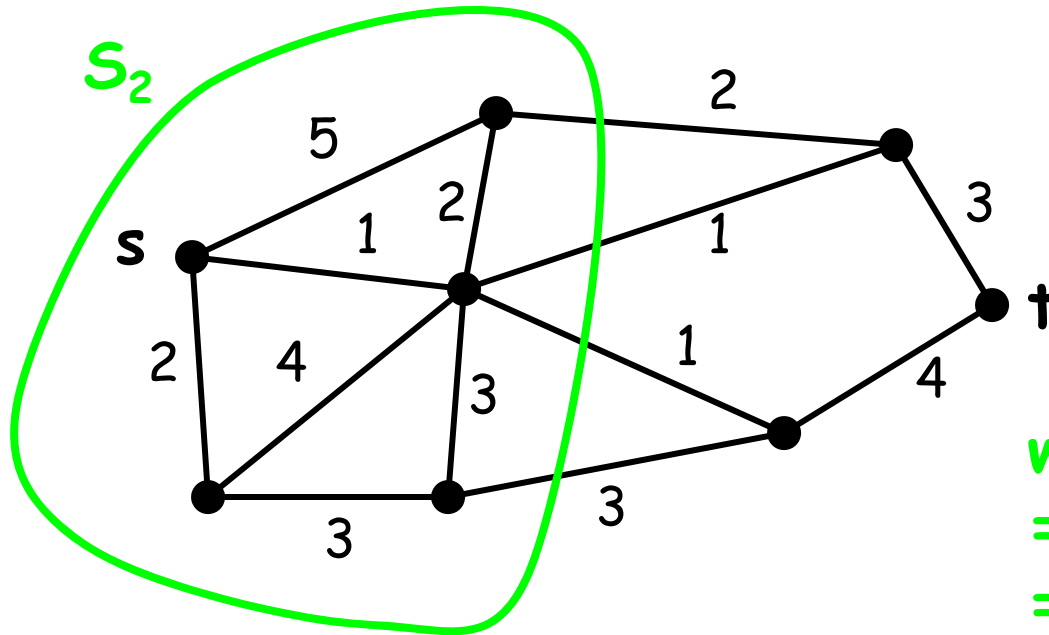
Recall: (s,t)-cut is a set  $S \subseteq V$  containing  $s$  but not  $t$ ; its weight is the sum of edge weights out of  $S$

# The problem: Counting minimum (s,t)-cuts

Input: a positively weighted (directed) planar graph  $G=(V,E)$  and two vertices  $s,t$

Output: the number of minimum (s,t)-cuts of  $G$

Example:



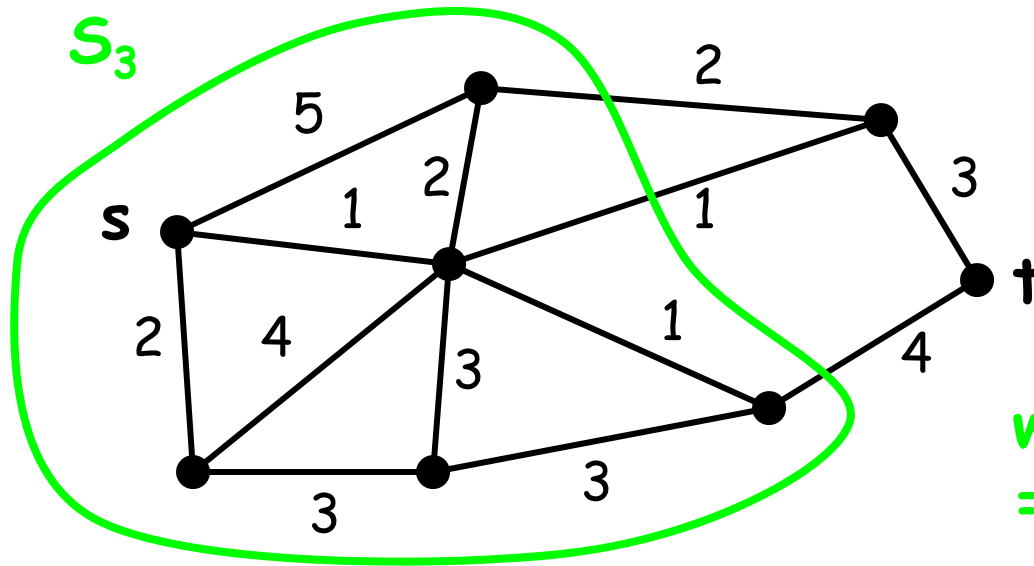
Recall: (s,t)-cut is a set  $S \subseteq V$  containing  $s$  but not  $t$ ; its weight is the sum of edge weights out of  $S$

# The problem: Counting minimum (s,t)-cuts

Input: a positively weighted (directed) planar graph  $G=(V,E)$  and two vertices  $s,t$

Output: the number of minimum (s,t)-cuts of  $G$

Example:



$$\begin{aligned} \text{weight}(S_3) &= 2+1+4 \\ &= 7 \end{aligned}$$

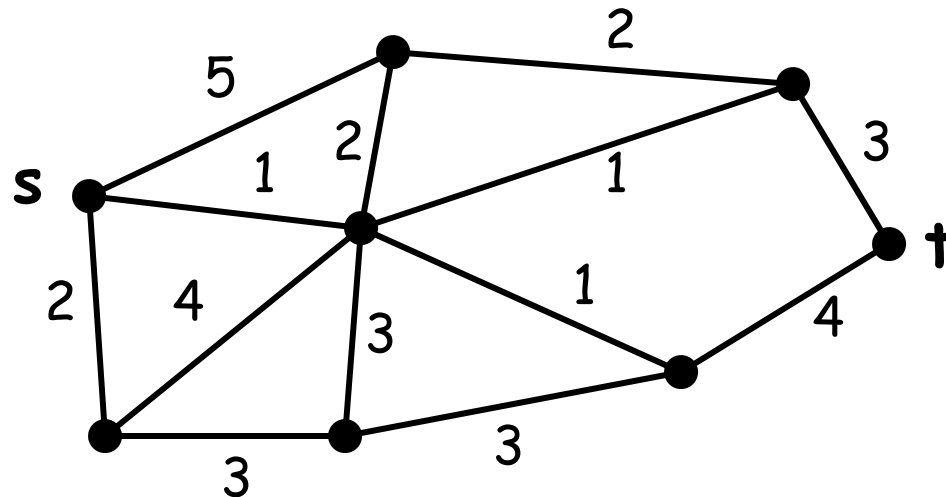
Recall: (s,t)-cut is a set  $S \subseteq V$  containing  $s$  but not  $t$ ; its weight is the sum of edge weights out of  $S$

# The problem: Counting minimum $(s,t)$ -cuts

Input: a positively weighted (directed) planar graph  $G=(V,E)$   
and two vertices  $s,t$

Output: the number of minimum  $(s,t)$ -cuts of  $G$

Example:



Min  $(s,t)$ -cut weight: 7

Number of min  $(s,t)$ -cuts: 5

# Motivation & related work

Initial motivation: **network reliability problems**

- number of min  $(s,t)$ -cuts useful in estimating the probability of disconnecting the network, e.g., [Colbourn '05]
- most related: Ball and Provan '82-'84
- [Ball and Provan '83]: efficient poly-time counting in (unweighted) planar (multi-) graphs when  $s,t$  on the same face
- [Nagamochi, Sun, Ibaraki '91]: unweighted multi-graphs, running time dependent on the number of cuts - not poly-time (improving Ball & Provan's run.time for non-planar graphs)

Other, recent, motivation: **computer vision**

- influential works, e.g.: [Boykov, Veksler & Zabih '01], [Boykov & Kolmogorov '04], [Geman & Geman '84]

# Motivation & related work

Other, recent, motivation: **computer vision**

- the simplest case: image segmentation where image represented by a planar graph
- user selects two points, the graph cut represents the segmentation
- currently in use only min-cut algorithms (optimization version), using an arbitrary min-cut
- many advantages of counting (and the related sampling) versions, e.g.:
  - user can choose from several cuts
  - can be used to compute the partition function that can be used to estimate model parameters

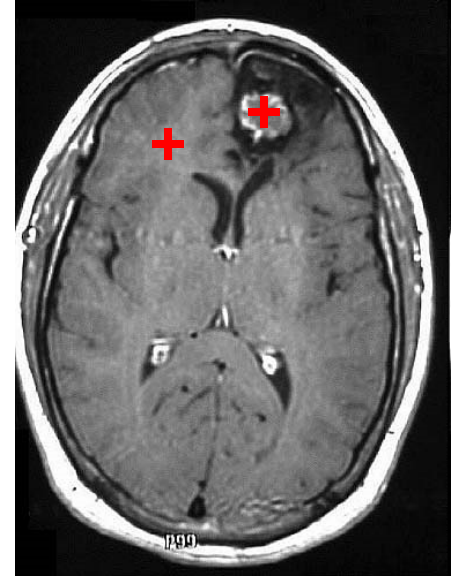


<http://path.upmc.edu/cases/case123.html>

# Motivation & related work

Other, recent, motivation: **computer vision**

- the simplest case: image segmentation where image represented by a planar graph
- user selects two points, the graph cut represents the segmentation
- currently in use only min-cut algorithms (optimization version), using an arbitrary min-cut
- many advantages of counting (and the related sampling) versions, e.g.:
  - user can choose from several cuts
  - can be used to compute the partition function that can be used to estimate model parameters



<http://path.upmc.edu/cases/case123.html>

# Our contributions

**Thm:** An  $O(nd + n \log n)$  algorithm computing the number of minimum  $(s,t)$ -cuts in weighted planar graphs, where:

- $n = \#$  vertices
- $d =$  length of the shortest  $s$ - $t$  path
- for directed graphs, assume all vertices are reachable from  $s$  and lead to  $t$ .

Comparison with earlier works:

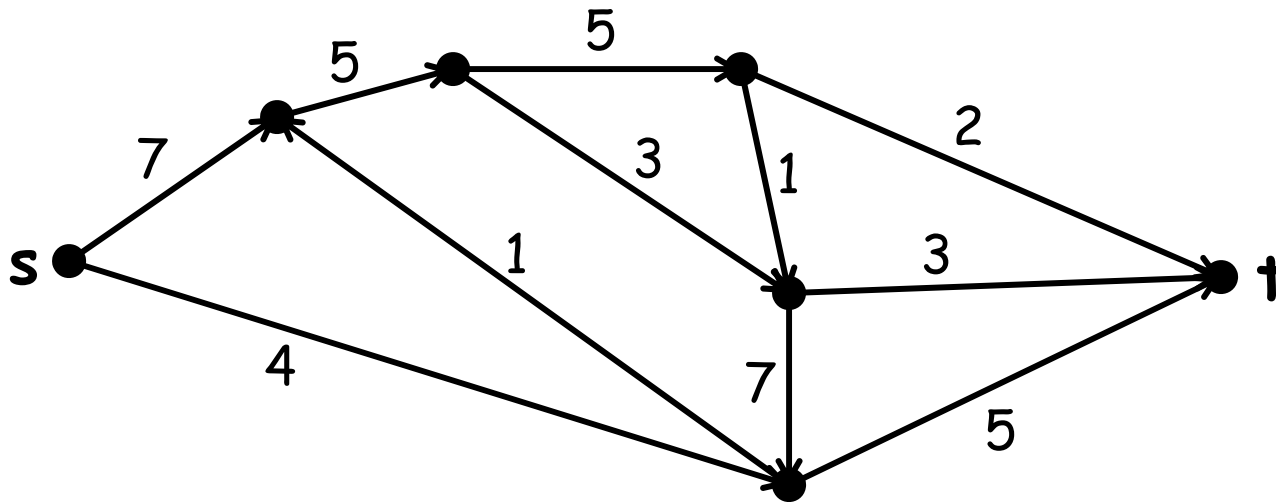
here no assumptions on the relative position of  $s$  and  $t$   
(and weighted)

-> important for computer vision applications

# Review of network flows

## Flow network:

- a directed graph with positive capacities on the edges, and
- two vertices  $s$  (the source) and  $t$  (the sink)



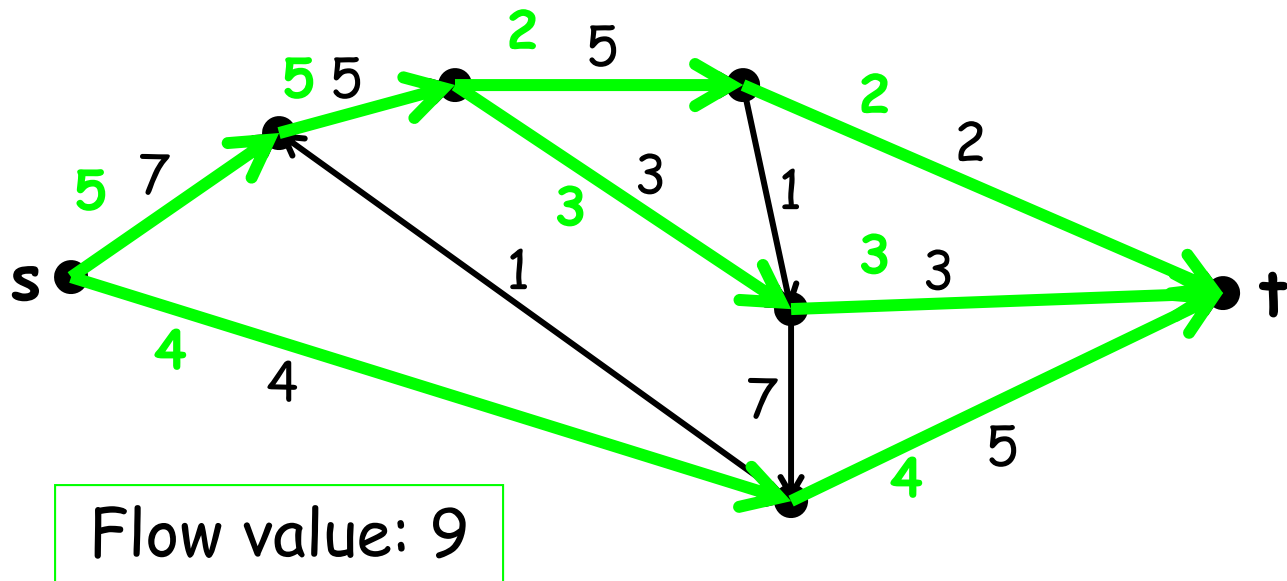
# Review of network flows

**Flow  $f$ :** flow amount on every edge satisfying:

- for every edge  $e$ : flow amount  $f(e) \leq$  capacity  $c(e)$ , and
- for every vertex  $v$  (except  $s, t$ ):

flow amount into  $v$  = flow amount out of  $v$

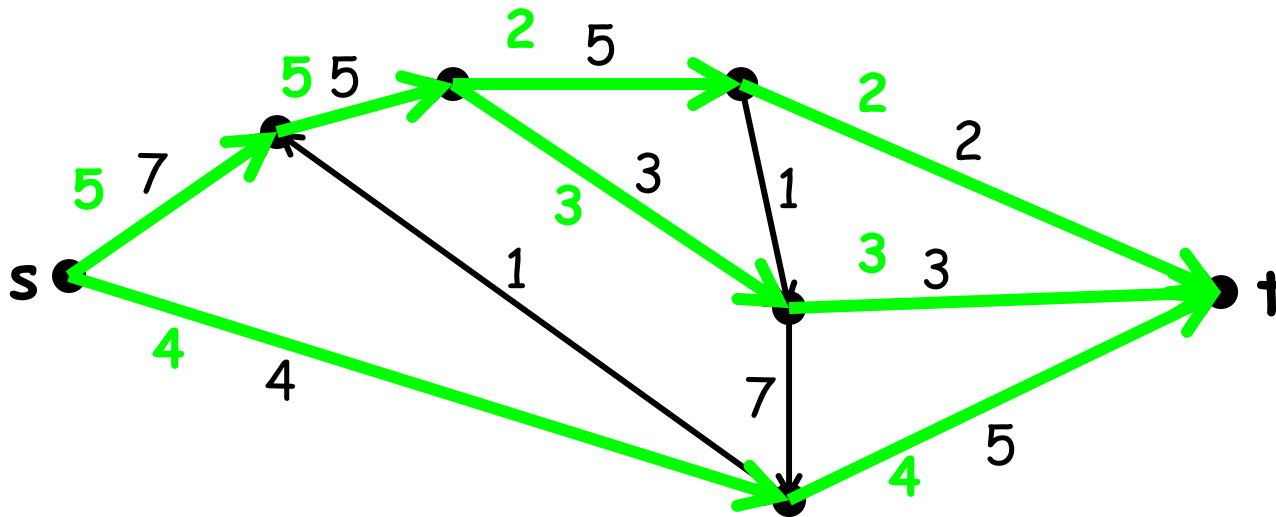
- flow value: amount out of  $s$  minus amount into  $s$



# Review of network flows

**Residual graph** of a flow  $f$ :

- **forward edges**: weight = capacity - flow
- **backward edges**: weight = flow

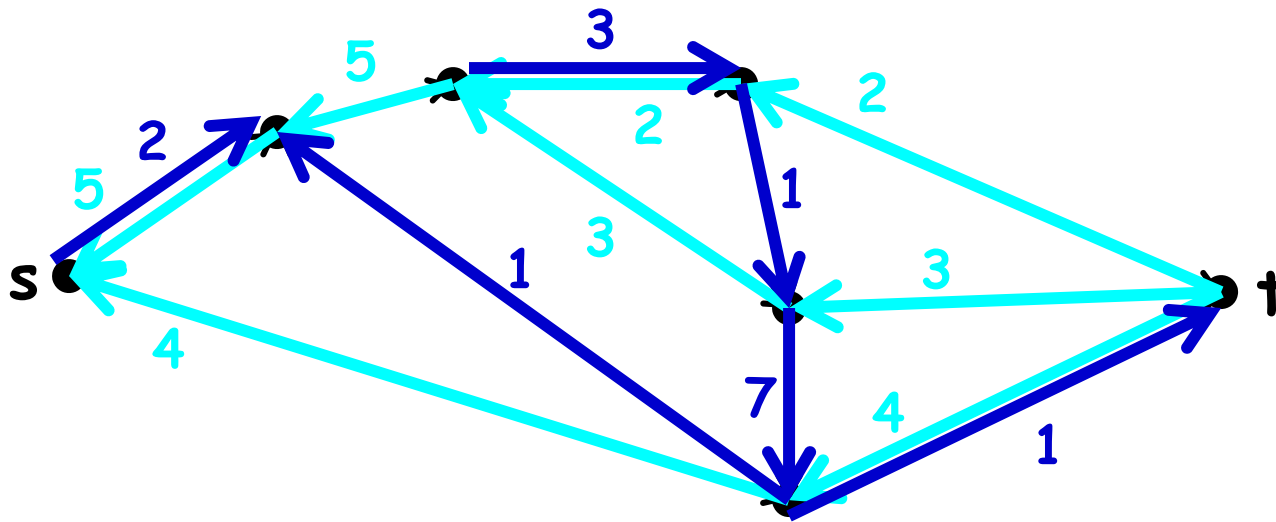




# Review of network flows

**Residual graph** of a flow  $f$ :

- **forward edges**: weight = capacity - flow
  - **backward edges**: weight = flow
- (only edges with positive weight)

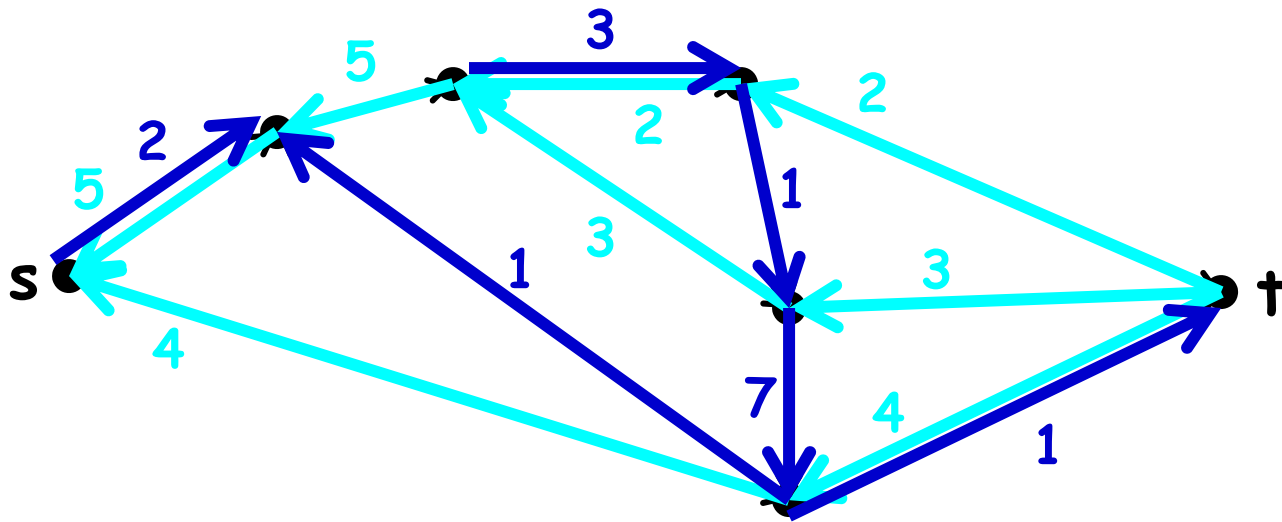


# Review of network flows

Ford-Fulkerson Thm:

value of max  $s$ - $t$  flow = value of min  $s$ - $t$  cut

Note: flow is max iff no  $s$ - $t$  path in the residual graph

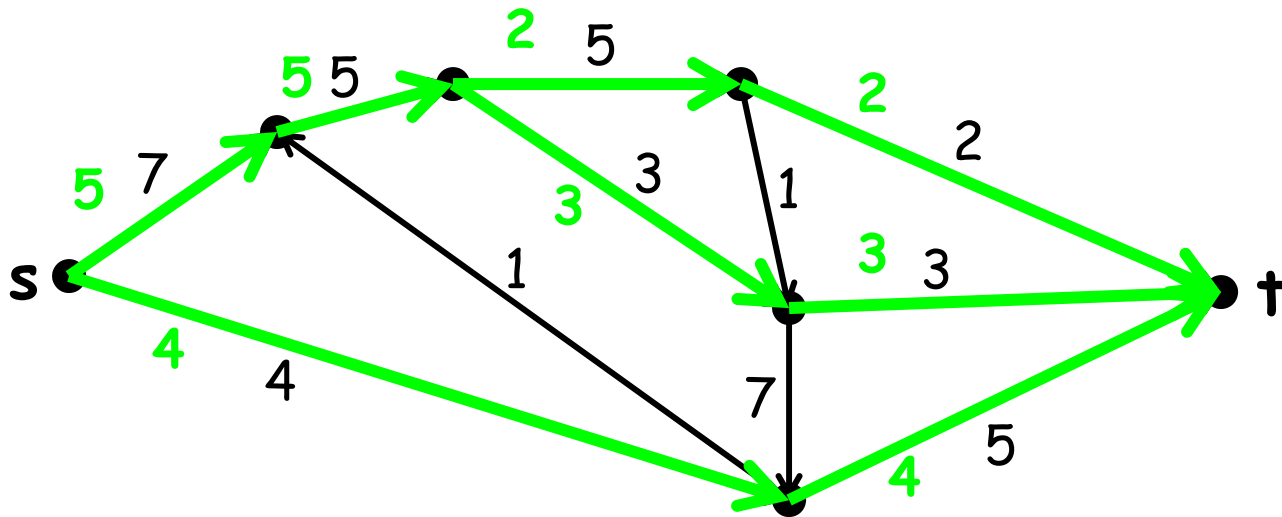


# Review of network flows

Ford-Fulkerson Thm:

value of max  $s$ - $t$  flow = value of min  $s$ - $t$  cut

Note: flow is max iff no  $s$ - $t$  path in the residual graph

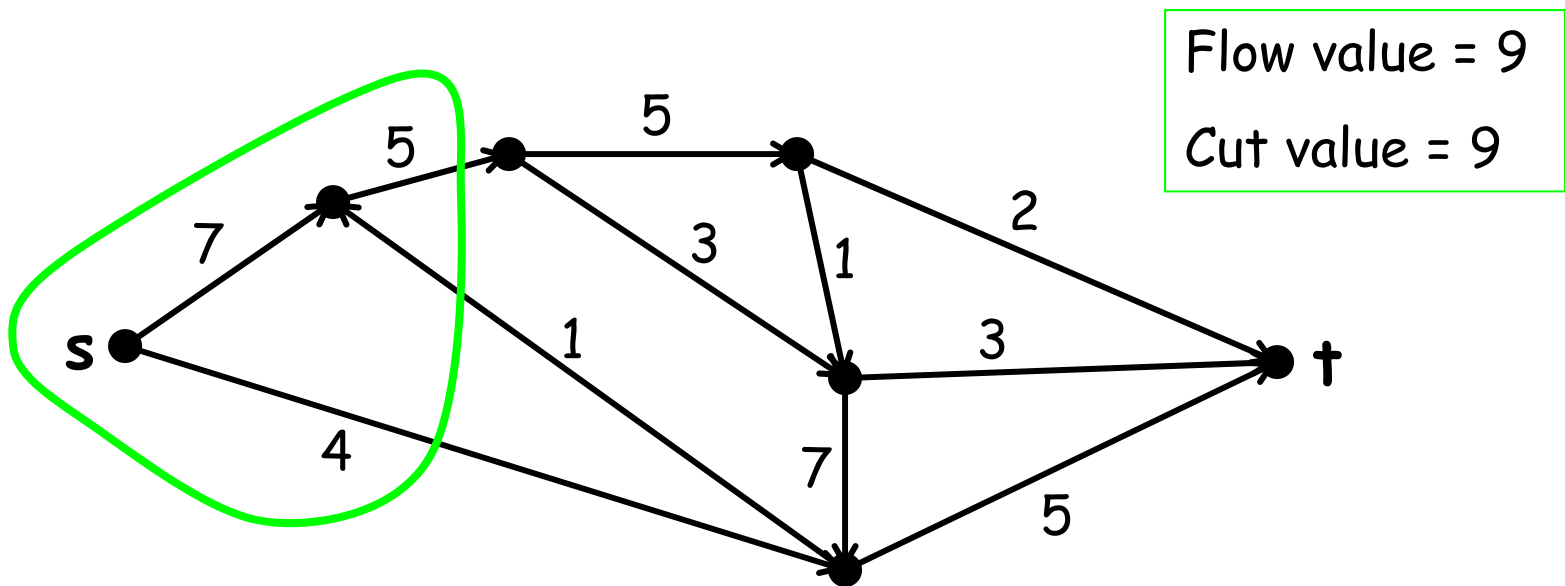


# Review of network flows

Ford-Fulkerson Thm:

value of max  $s$ - $t$  flow = value of min  $s$ - $t$  cut

Note: flow is max iff no  $s$ - $t$  path in the residual graph

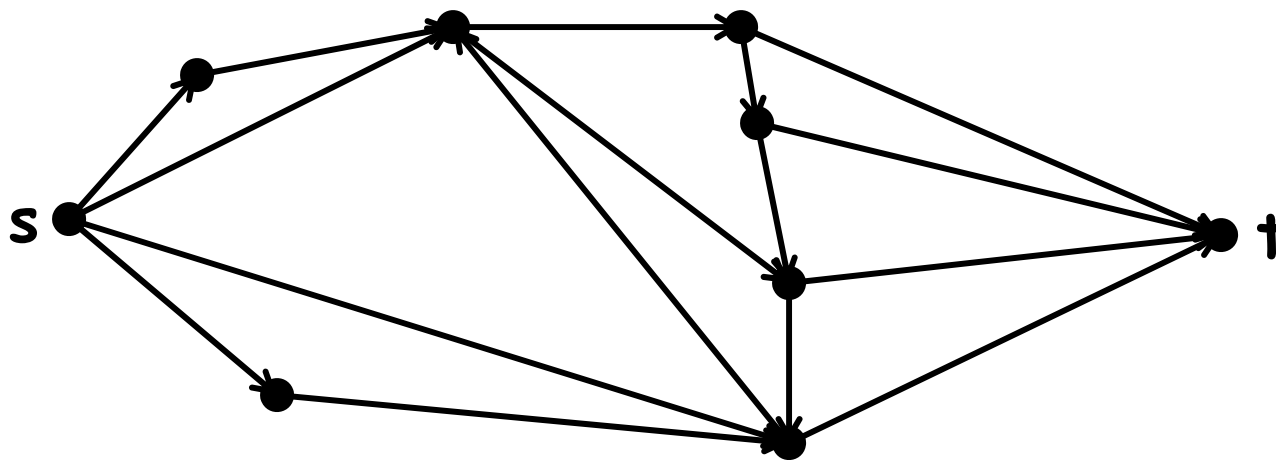


# Ball & Provan's counting approach

Given an unweighted (multi-)graph:

1. Find a max flow and construct the residual graph
2. Contract strongly connected components
3. Compute # "forward-cuts" in the DAG

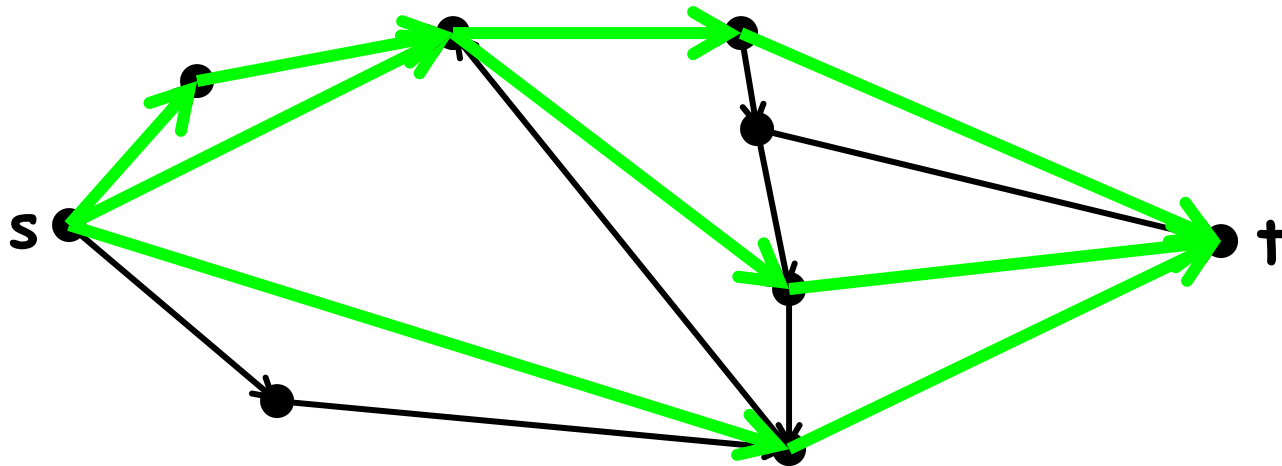
(forward-cuts = maximal antichains in the poset)



# Ball & Provan's counting approach

Given an unweighted (multi-)graph:

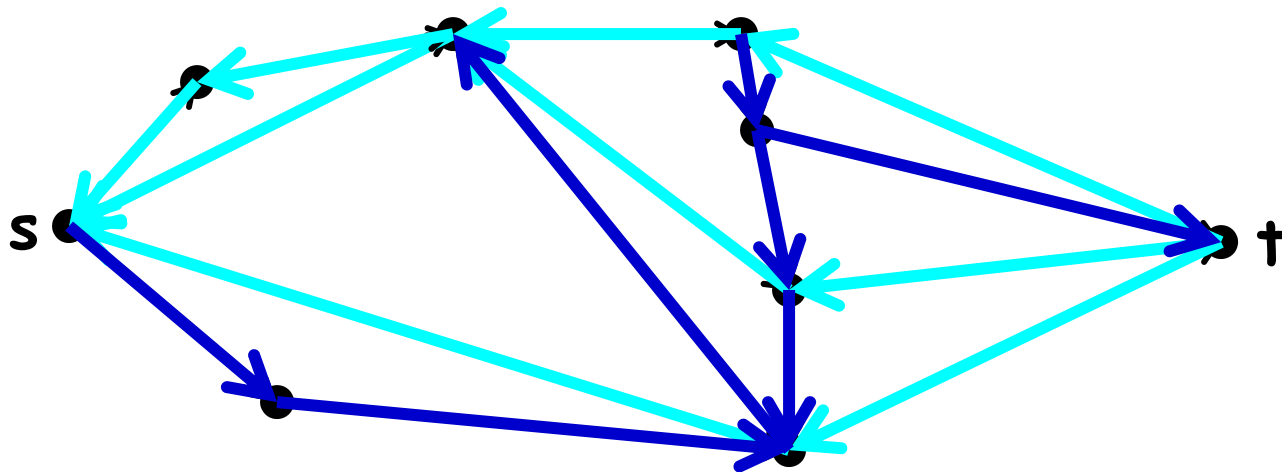
1. Find a max flow and construct the residual graph
2. Contract strongly connected components
3. Compute # "forward-cuts" in the DAG



# Ball & Provan's counting approach

Given an unweighted (multi-)graph:

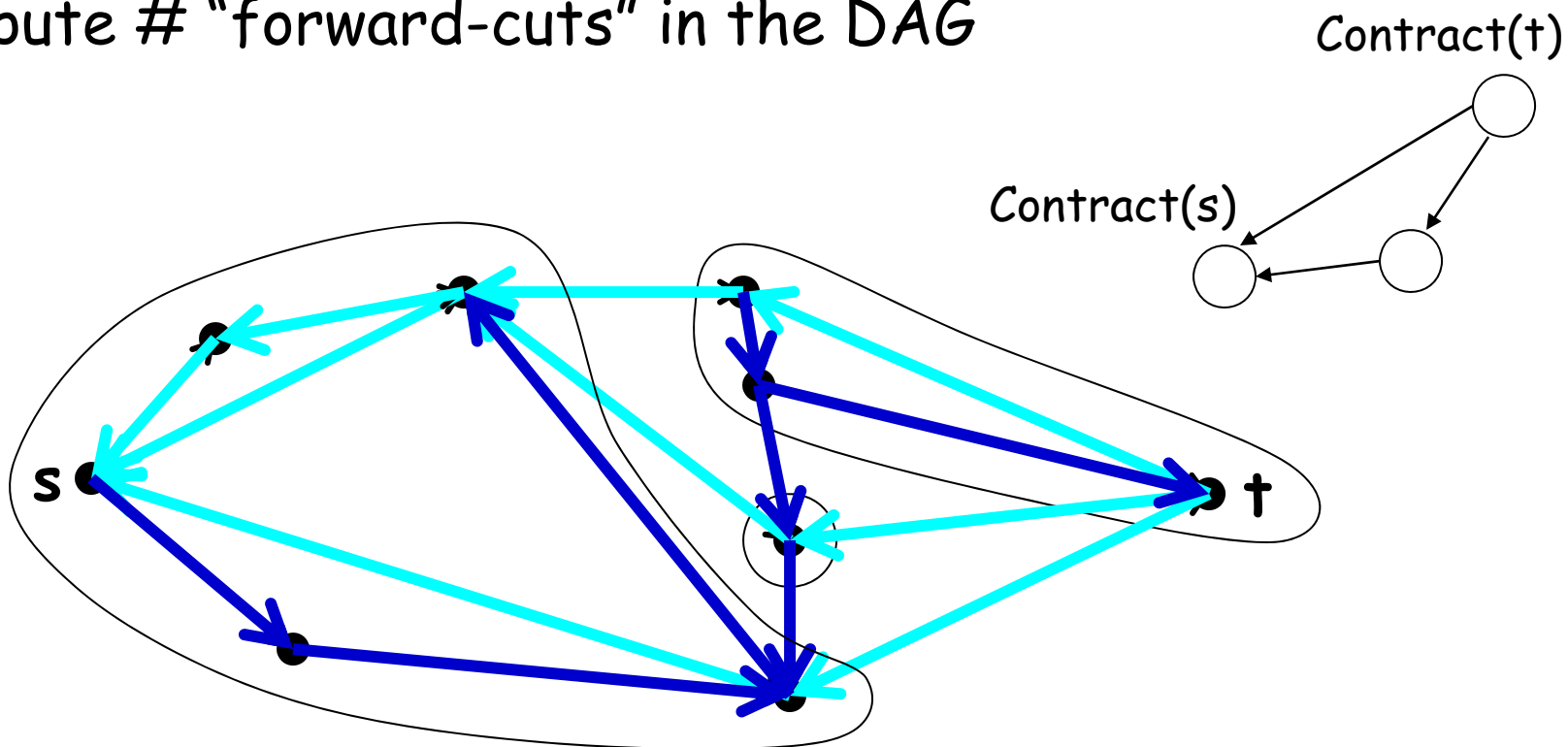
1. Find a max flow and construct the residual graph
2. Contract strongly connected components
3. Compute # "forward-cuts" in the DAG



# Ball & Provan's counting approach

Given an unweighted (multi-)graph:

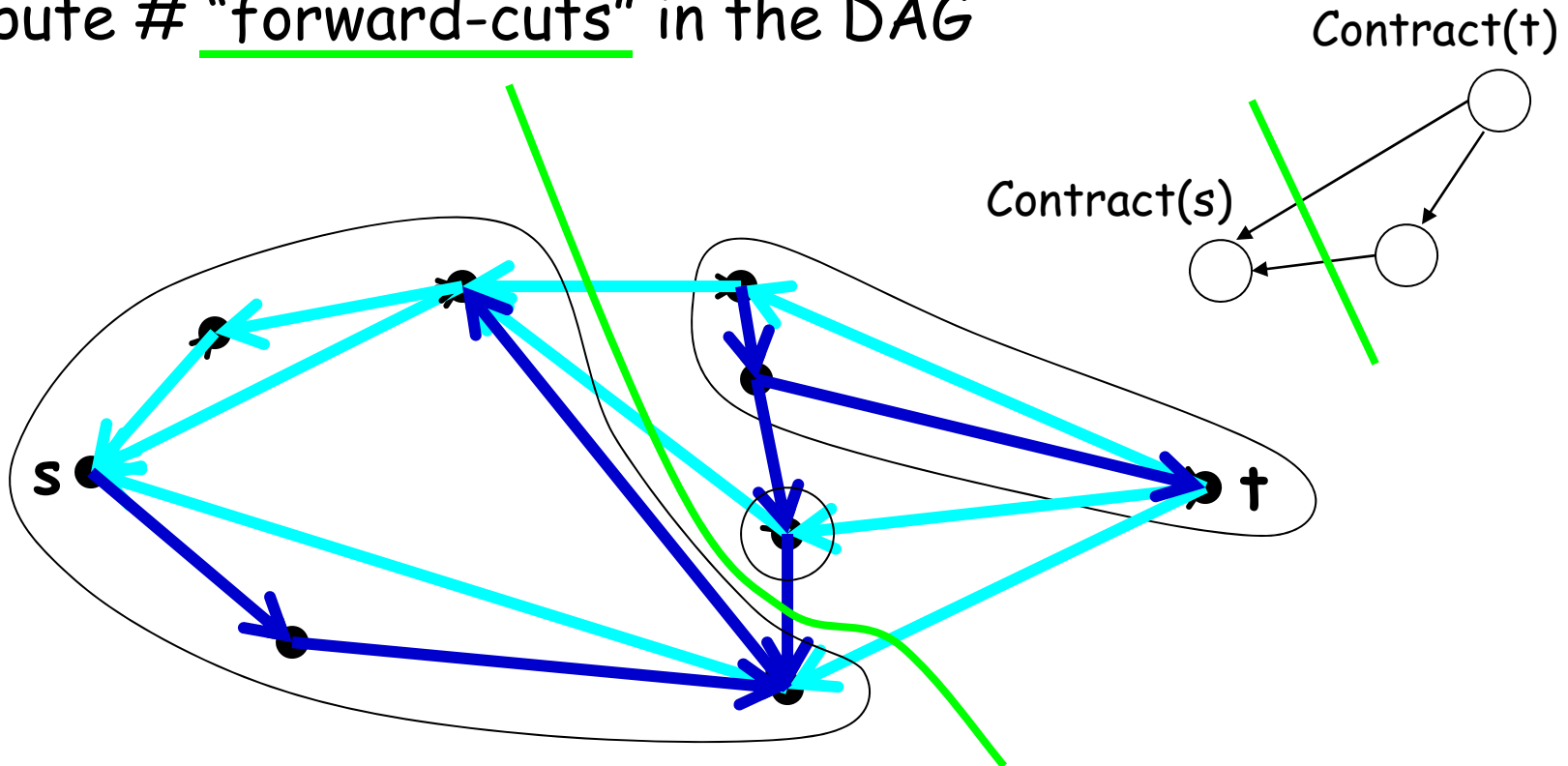
1. Find a max flow and construct the residual graph
2. Contract strongly connected components
3. Compute # "forward-cuts" in the DAG



# Ball & Provan's counting approach

Given an unweighted (multi-)graph:

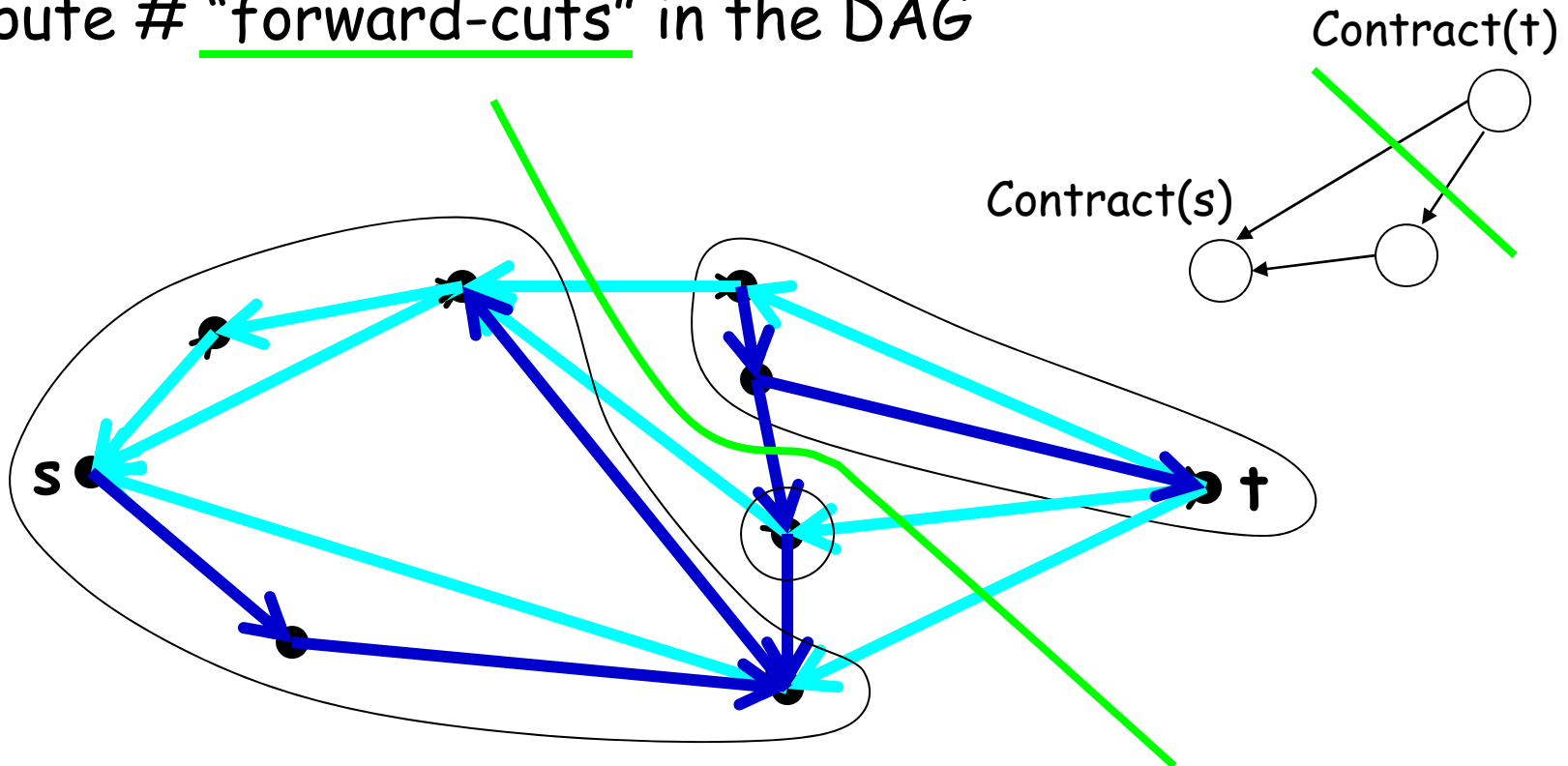
1. Find a max flow and construct the residual graph
2. Contract strongly connected components
3. Compute # "forward-cuts" in the DAG



# Ball & Provan's counting approach

Given an unweighted (multi-)graph:

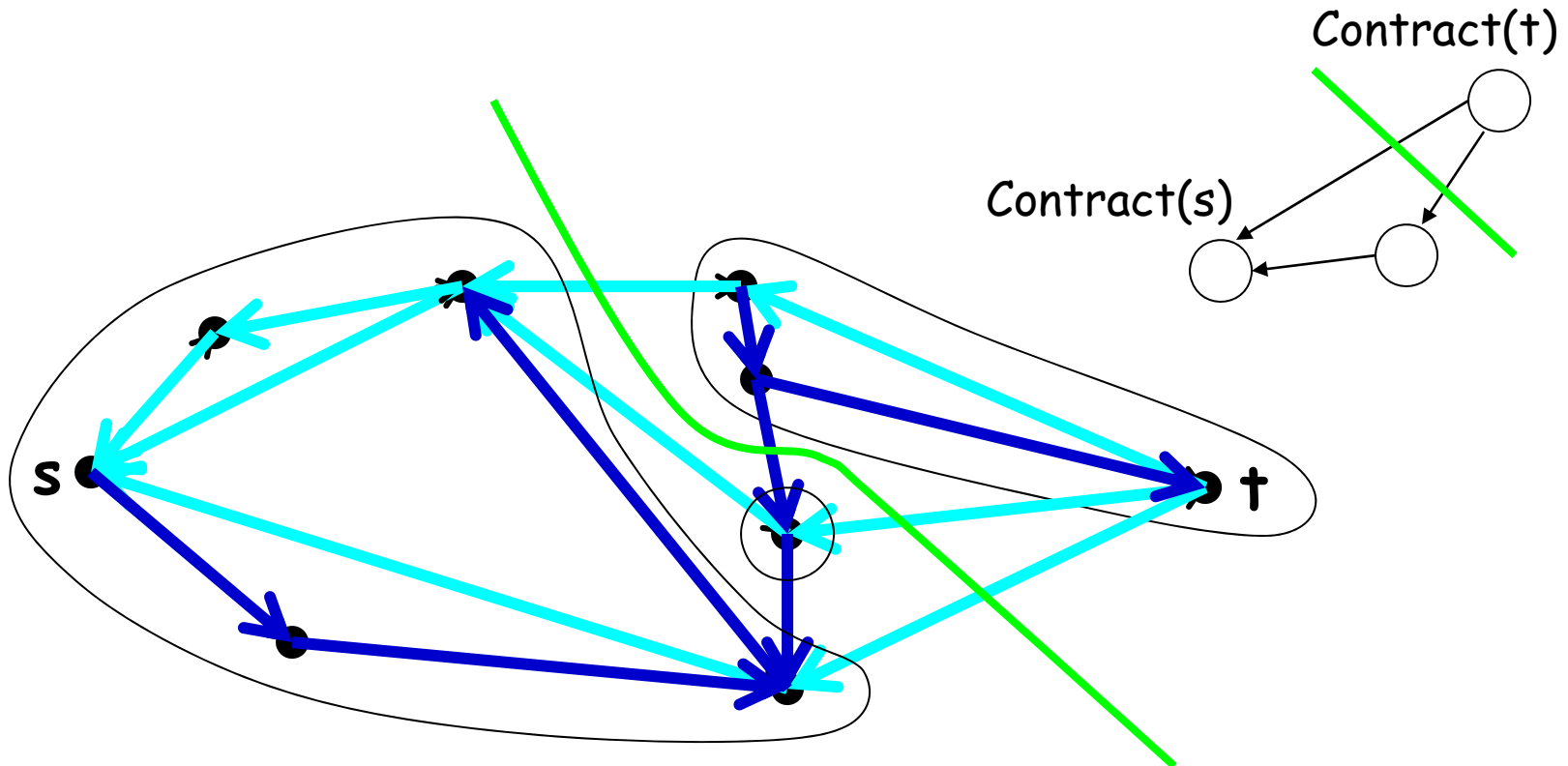
1. Find a max flow and construct the residual graph
2. Contract strongly connected components
3. Compute # "forward-cuts" in the DAG



# Ball & Provan's counting approach

"Forward-cut:" a set of vertices  $S$  such that:

- contains  $\text{Contract}(s)$  and not  $\text{Contract}(t)$ , and
- for every vertex in  $S$ , all successors also in  $S$

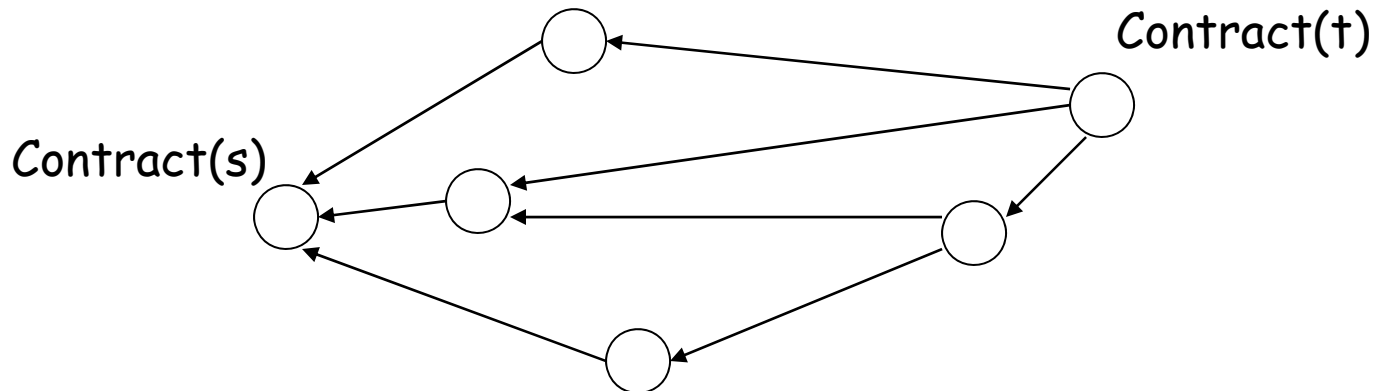


# Ball & Provan's counting approach

"Forward-cut:" a set of vertices  $S$  such that:

- contains  $\text{Contract}(s)$  and not  $\text{Contract}(t)$ , and
- for every vertex in  $S$ , all successors also in  $S$

Another DAG example:

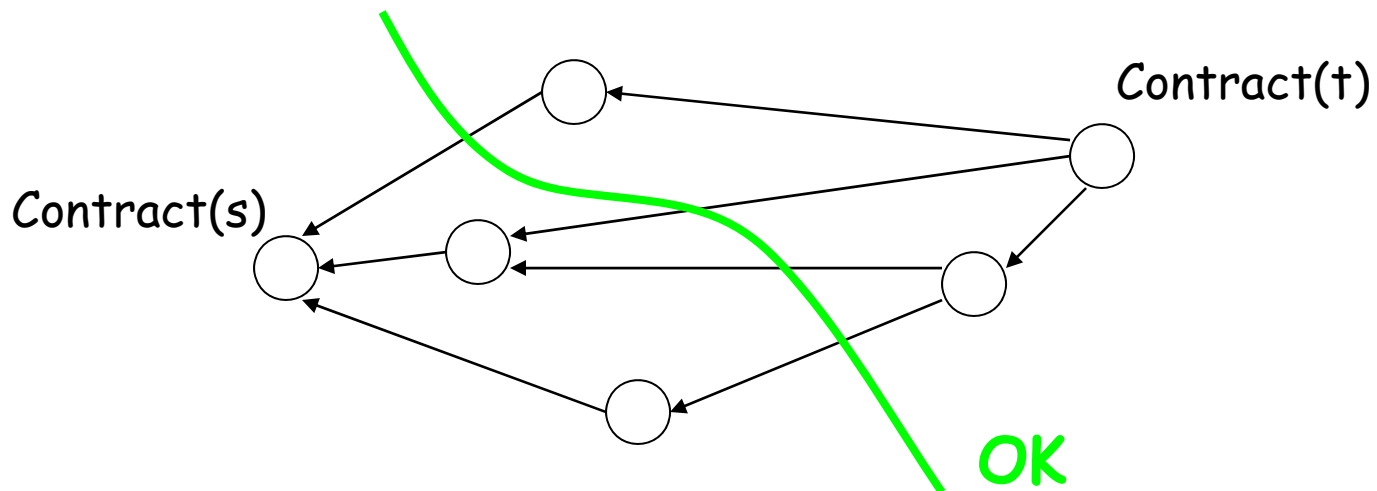


# Ball & Provan's counting approach

"Forward-cut:" a set of vertices  $S$  such that:

- contains  $\text{Contract}(s)$  and not  $\text{Contract}(t)$ , and
- for every vertex in  $S$ , all successors also in  $S$

Another DAG example:

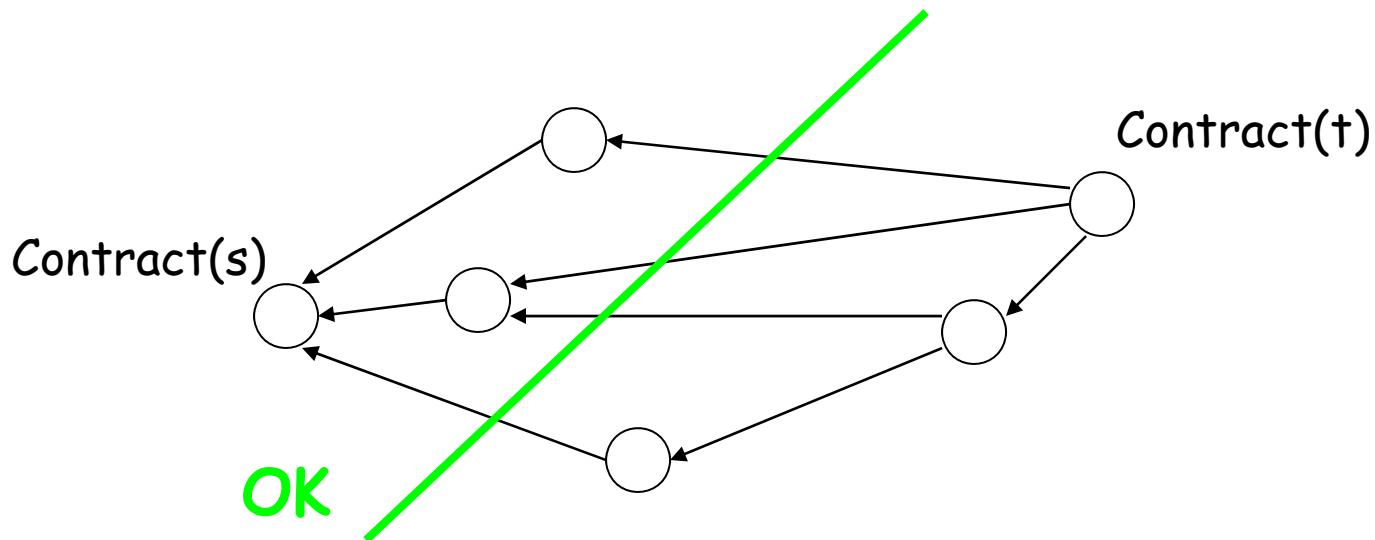


# Ball & Provan's counting approach

"Forward-cut:" a set of vertices  $S$  such that:

- contains  $\text{Contract}(s)$  and not  $\text{Contract}(t)$ , and
- for every vertex in  $S$ , all successors also in  $S$

Another DAG example:

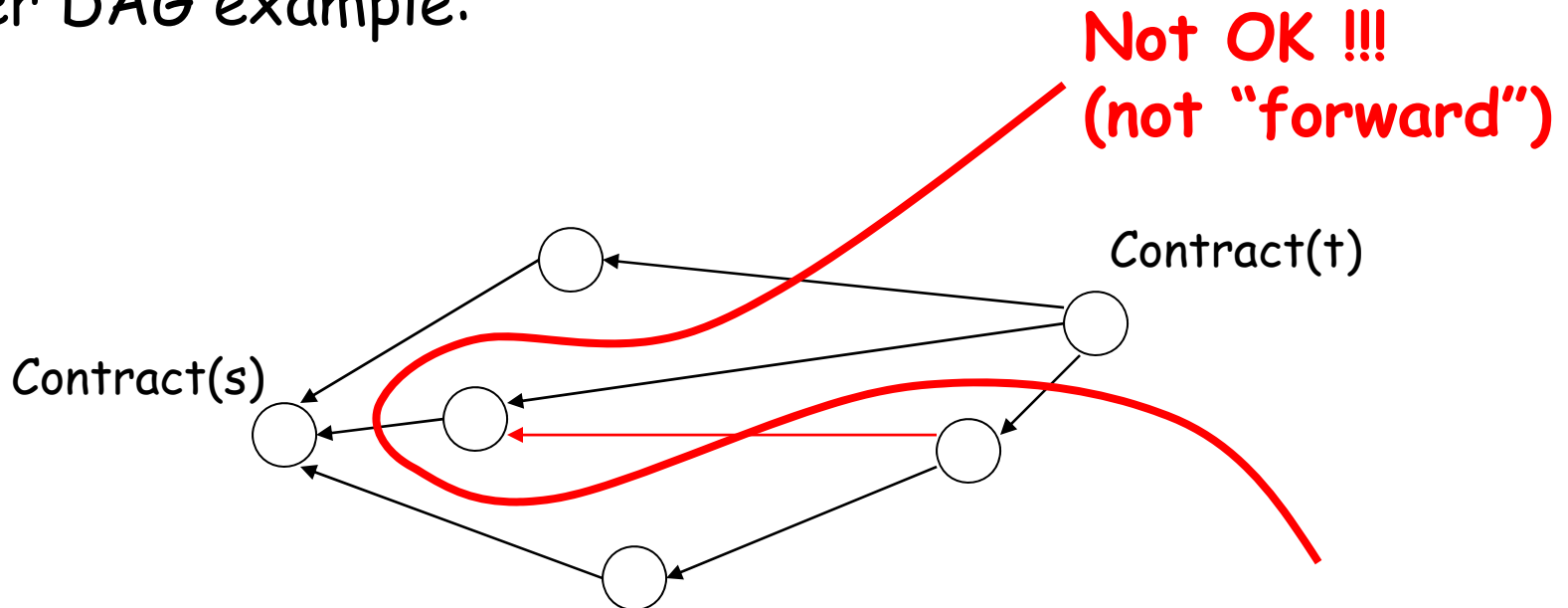


# Ball & Provan's counting approach

"Forward-cut:" a set of vertices  $S$  such that:

- contains  $\text{Contract}(s)$  and not  $\text{Contract}(t)$ , and
- for every vertex in  $S$ , all successors also in  $S$

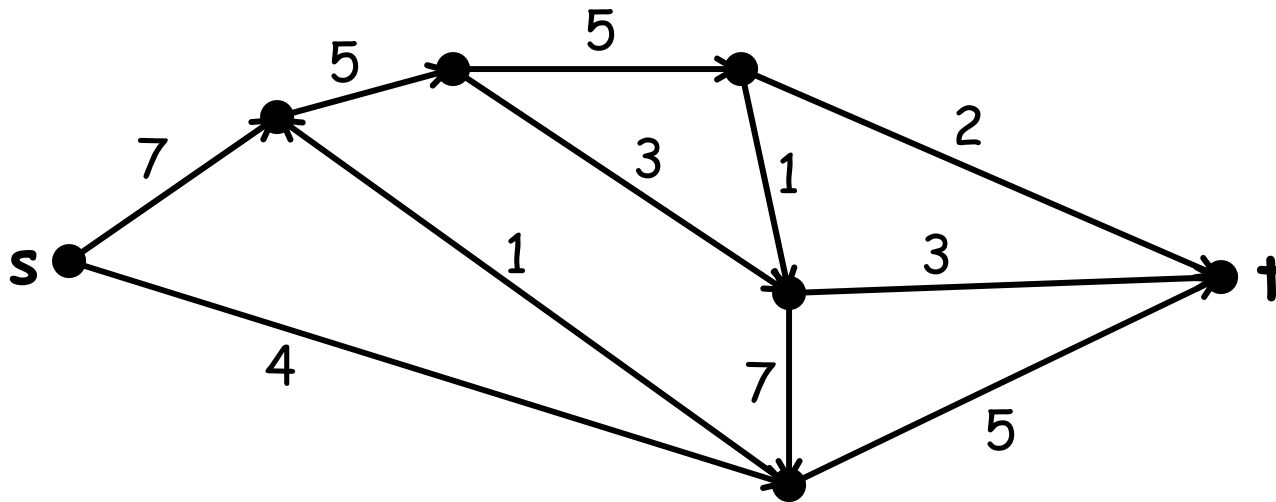
Another DAG example:



# The weighted case

What if the graph is (positively) weighted ?

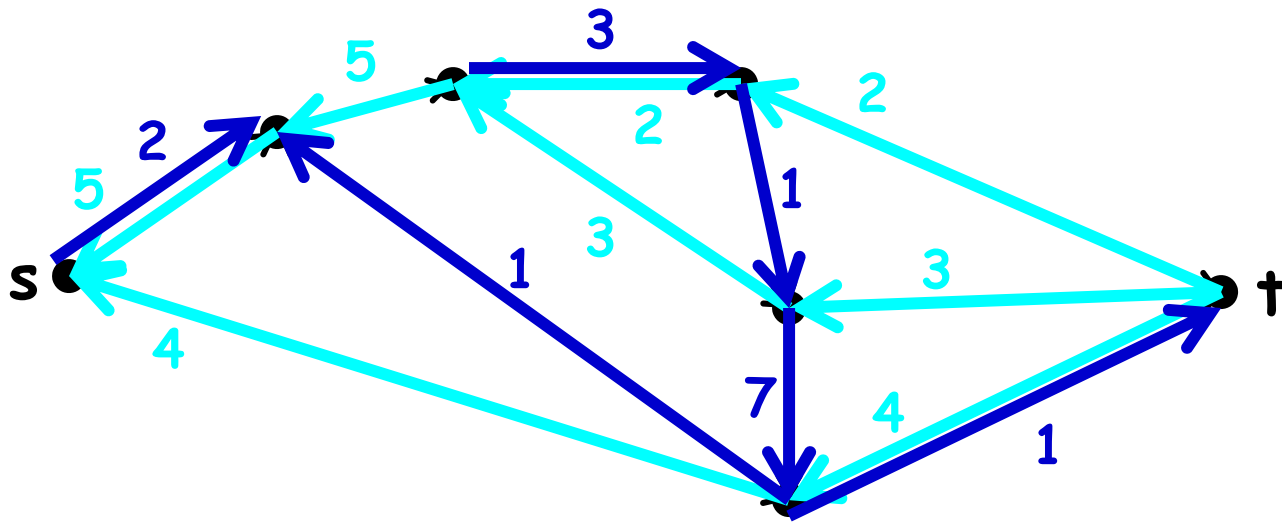
- the reduction to forward-cuts still works [\[this paper\]](#) (note: the resulting DAG is unweighted)
- also note: the reduction works for all graphs, not just planar graphs



# The weighted case

What if the graph is (positively) weighted ?

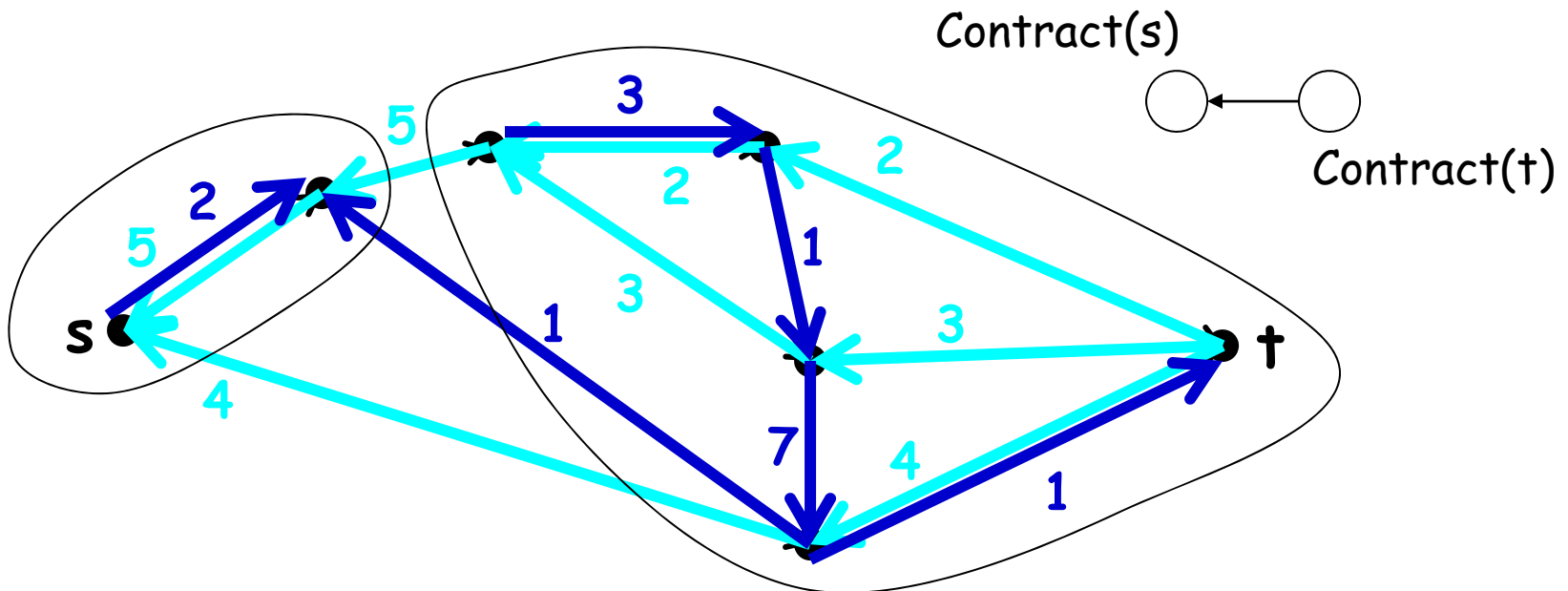
- the reduction to forward-cuts still works [\[this paper\]](#) (note: the resulting DAG is unweighted)
- also note: the reduction works for all graphs, not just planar graphs



# The weighted case

What if the graph is (positively) weighted?

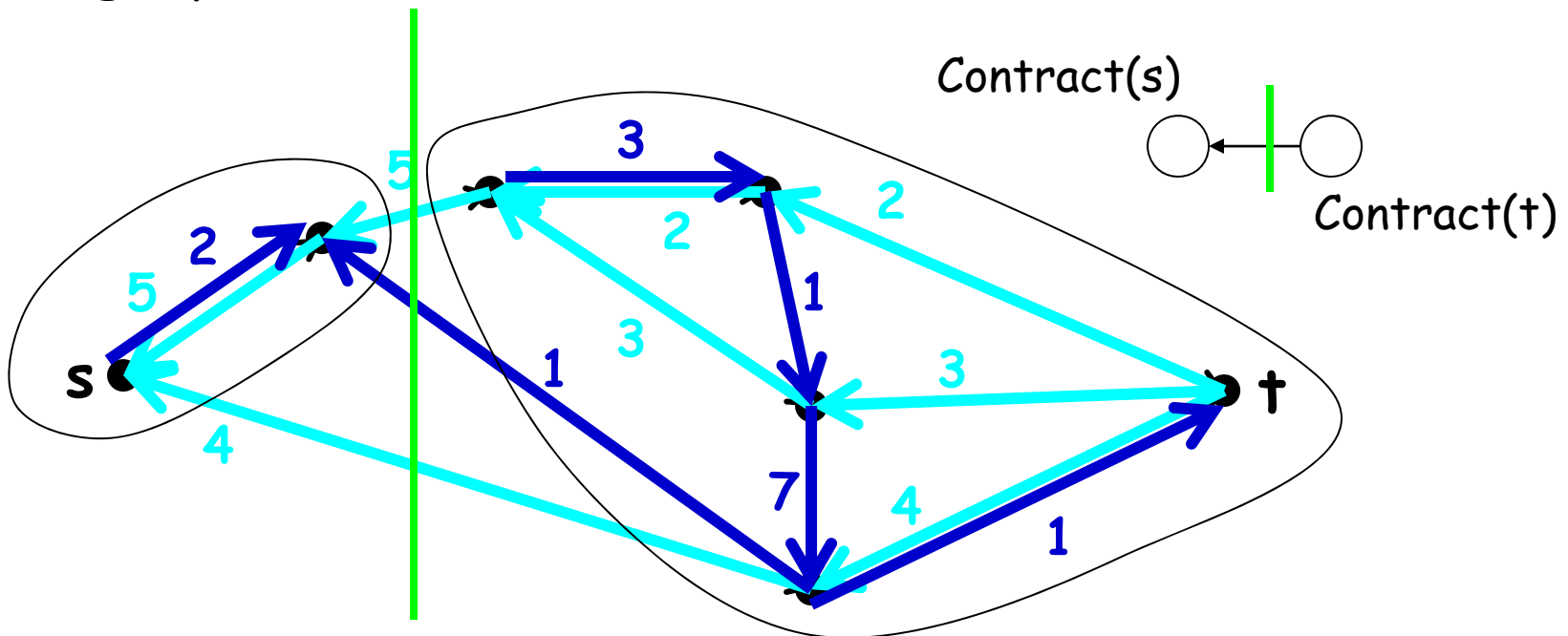
- the reduction to forward-cuts still works [this paper] (note: the resulting DAG is unweighted)
- also note: the reduction works for all graphs, not just planar graphs



# The weighted case

What if the graph is (positively) weighted?

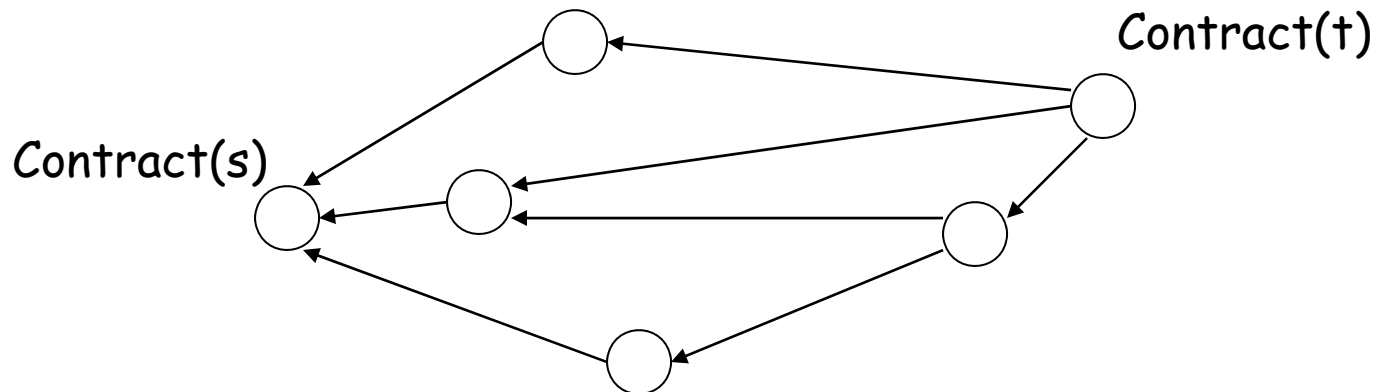
- the reduction to forward-cuts still works [this paper] (note: the resulting DAG is unweighted)
- also note: the reduction works for all graphs, not just planar graphs



# "Forward-cuts" in planar DAGs

Observe: Planar input graph  $\rightarrow$  planar DAG

Goal: count "forward-cuts" (or maximal antichains)



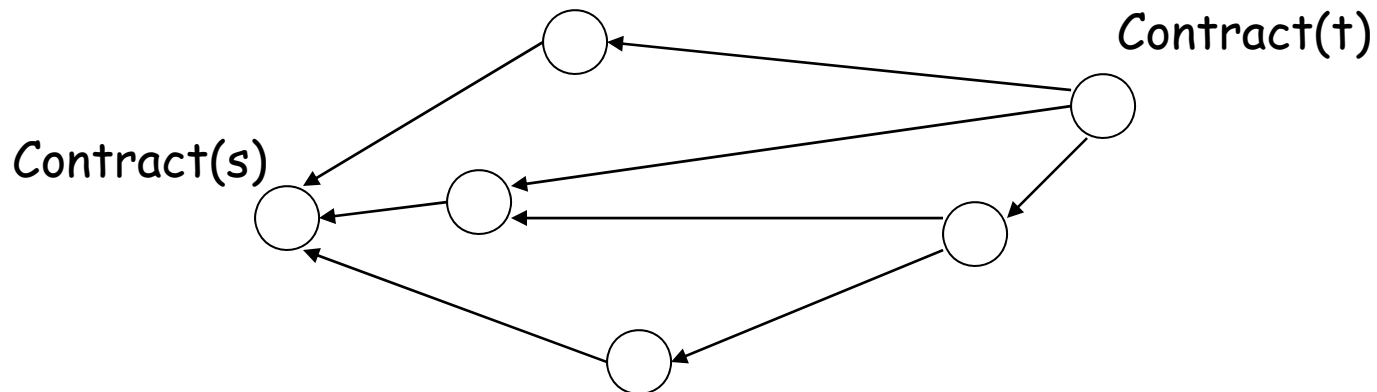
# “Forward-cuts” in planar DAGs

Observe: Planar input graph  $\rightarrow$  planar DAG

Goal: count “forward-cuts” (or maximal antichains)

Ball & Provan’s algorithm for both  $s, t$  on the outer face:

- split the outer face into the “top” and the “bottom” face
- count all “top”-“bottom” paths in the dual graph



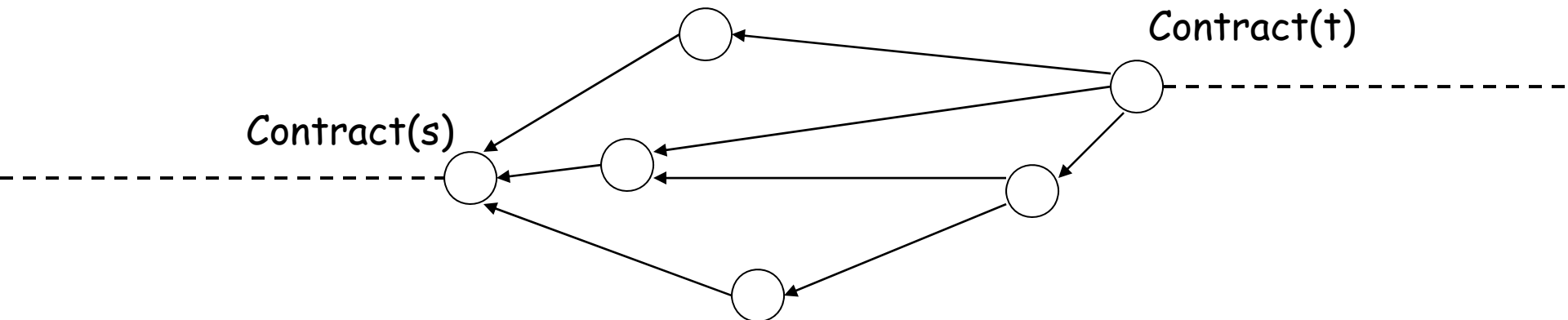
# “Forward-cuts” in planar DAGs

Observe: Planar input graph  $\rightarrow$  planar DAG

Goal: count “forward-cuts” (or maximal antichains)

Ball & Provan’s algorithm for both  $s, t$  on the outer face:

- split the outer face into the “top” and the “bottom” face
- count all “top”-“bottom” paths in the dual graph



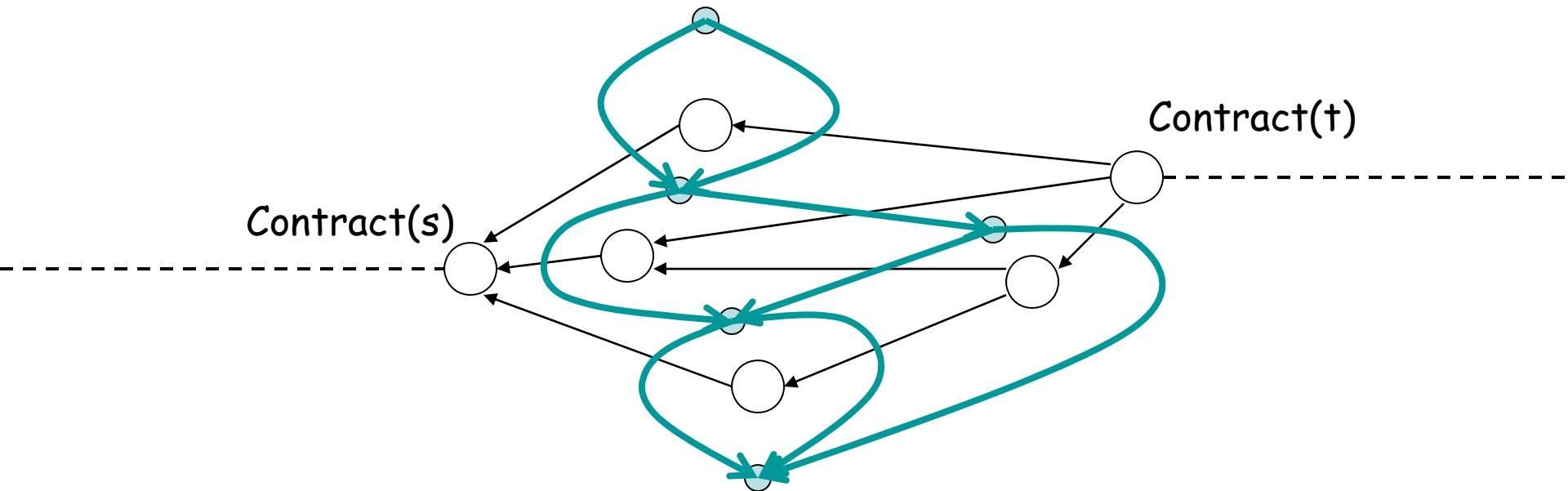
# "Forward-cuts" in planar DAGs

Observe: Planar input graph  $\rightarrow$  planar DAG

Goal: count "forward-cuts" (or maximal antichains)

Ball & Provan's algorithm for both  $s, t$  on the outer face:

- split the outer face into the "top" and the "bottom" face
- count all "top"-"bottom" paths in the dual graph



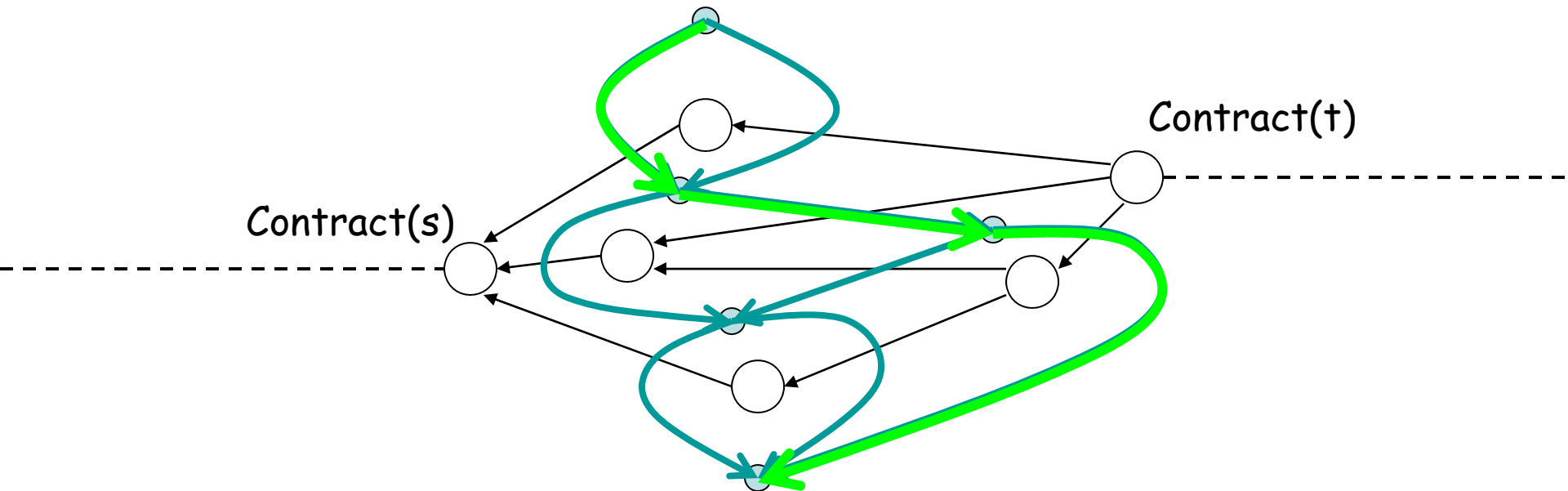
# "Forward-cuts" in planar DAGs

Observe: Planar input graph  $\rightarrow$  planar DAG

Goal: count "forward-cuts" (or maximal antichains)

Ball & Provan's algorithm for both  $s, t$  on the outer face:

- split the outer face into the "top" and the "bottom" face
- count all "top"- "bottom" paths in the dual graph



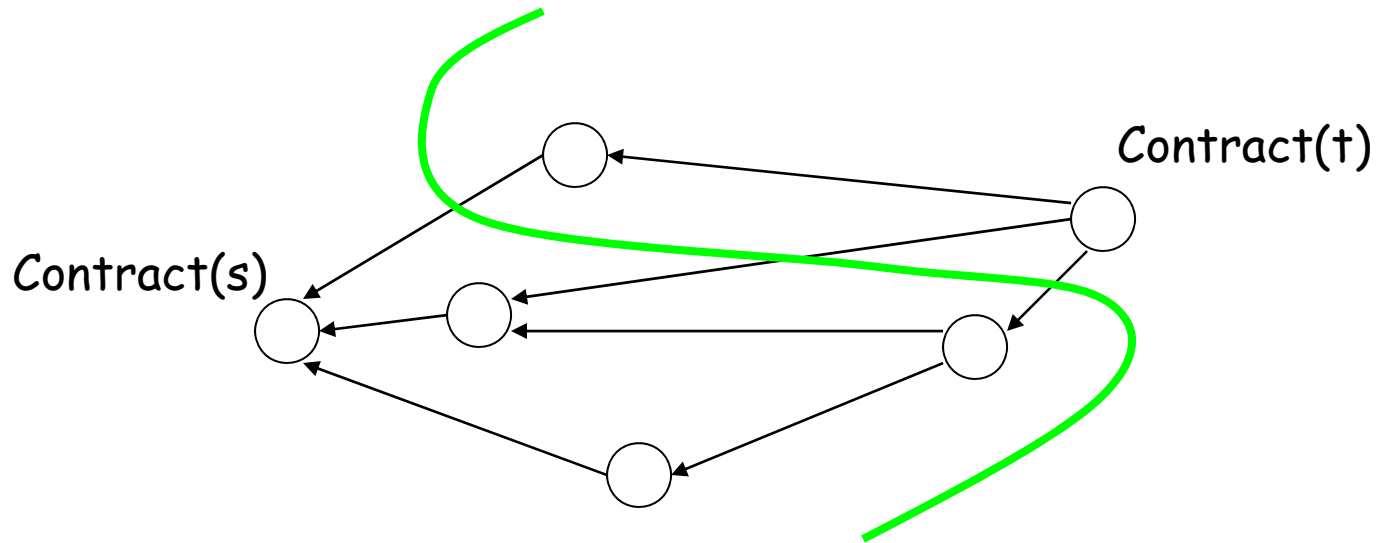
# “Forward-cuts” in planar DAGs

Observe: Planar input graph  $\rightarrow$  planar DAG

Goal: count “forward-cuts” (or maximal antichains)

Ball & Provan’s algorithm for both  $s, t$  on the outer face:

- split the outer face into the “top” and the “bottom” face
- count all “top”-“bottom” paths in the dual graph



# “Forward-cuts” in planar DAGs

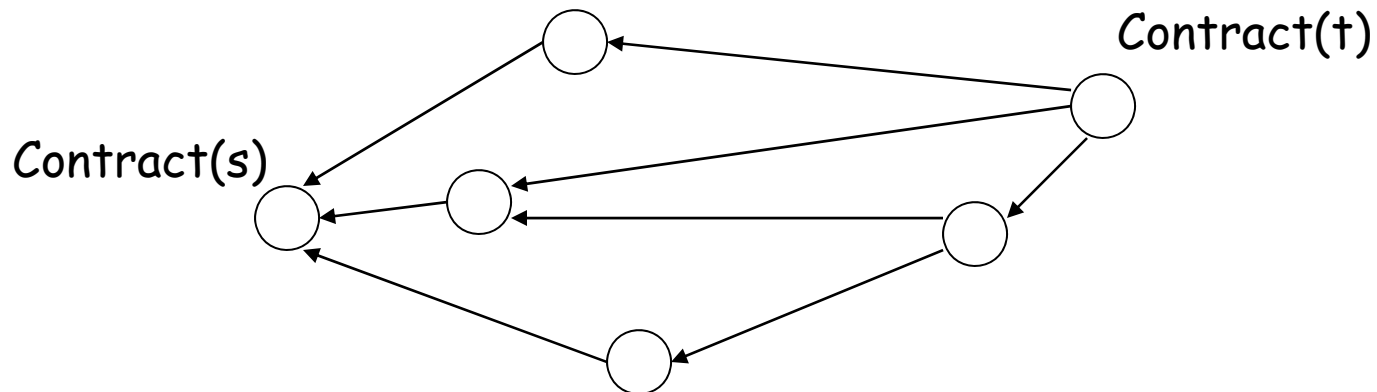
Observe: Planar input graph  $\rightarrow$  planar DAG

Goal: count “forward-cuts” (or maximal antichains)

Ball & Provan’s algorithm for both  $s, t$  on the outer face:

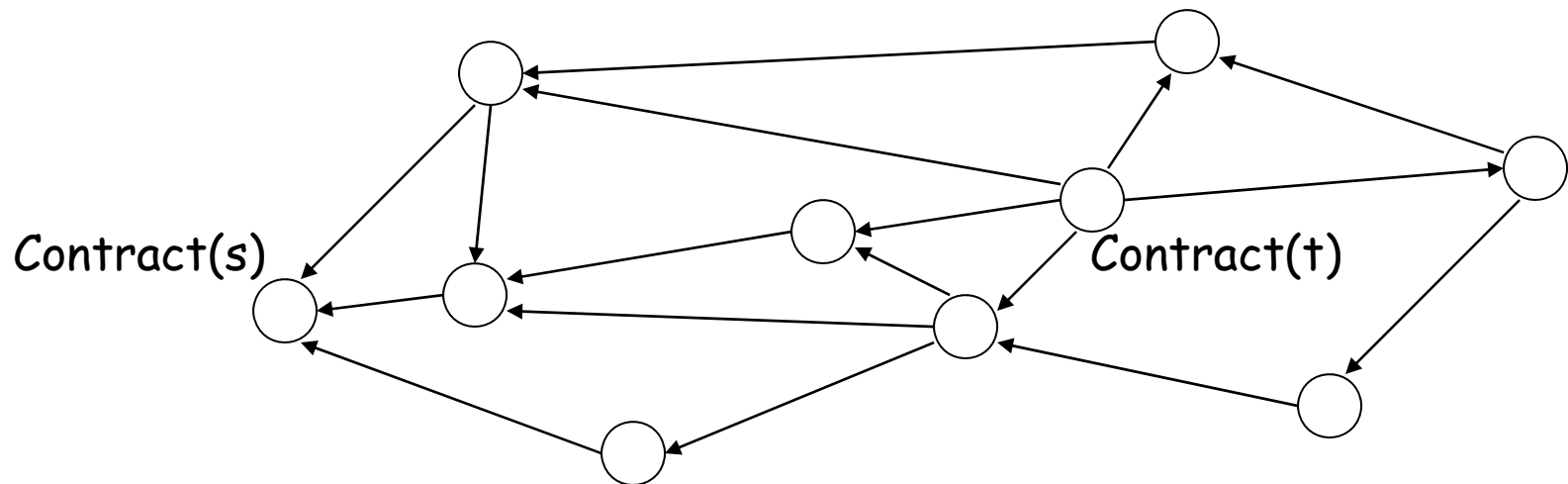
- split the outer face into the “top” and the “bottom” face
- count all “top”-“bottom” paths in the dual graph

Note: poly-time because the dual is a DAG



# “Forward-cuts” and different faces

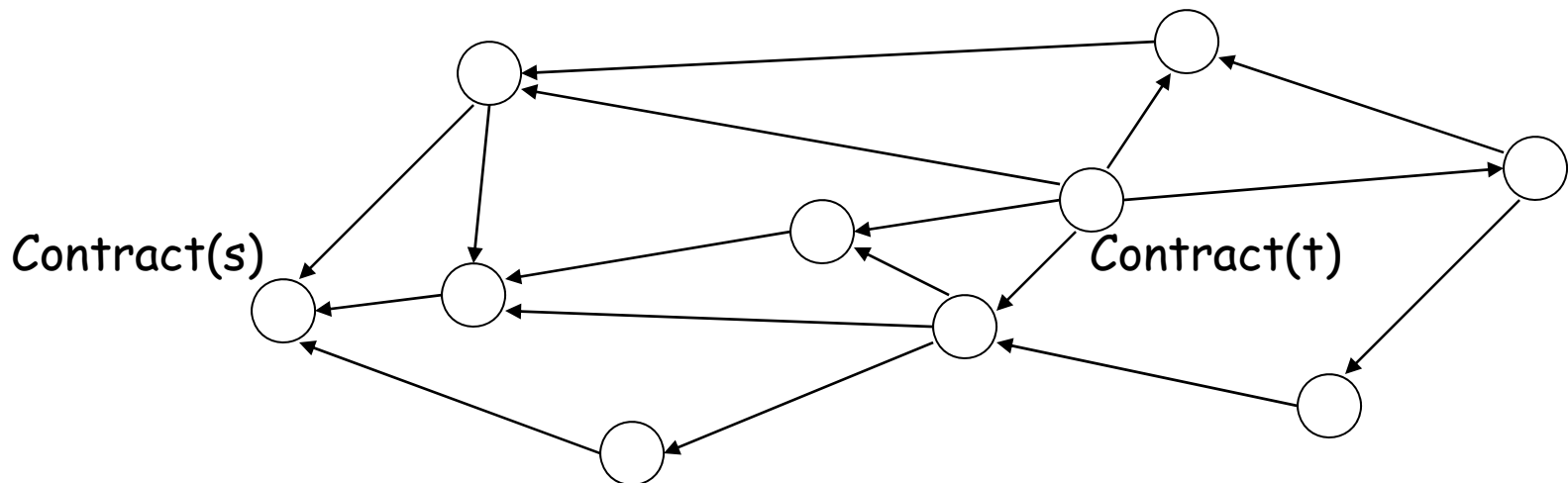
What if  $s, t$  are on different faces? [\[this paper\]](#)



# “Forward-cuts” and different faces

What if  $s, t$  are on different faces ? [\[this paper\]](#)

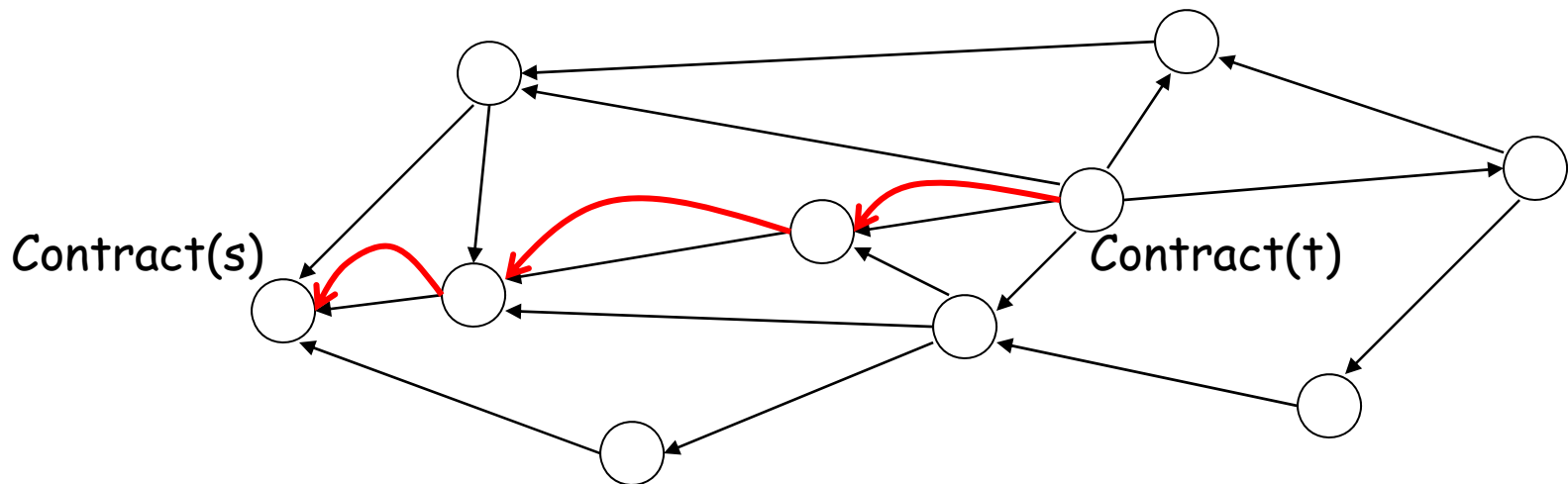
- find  $\text{contract}(t)$ - $\text{contract}(s)$  path, duplicate edges, “merge” edges within the same face
- construct the dual, except no edges cross the new path
- sum # paths between faces sharing a new edge



# “Forward-cuts” and different faces

What if  $s, t$  are on different faces ? [\[this paper\]](#)

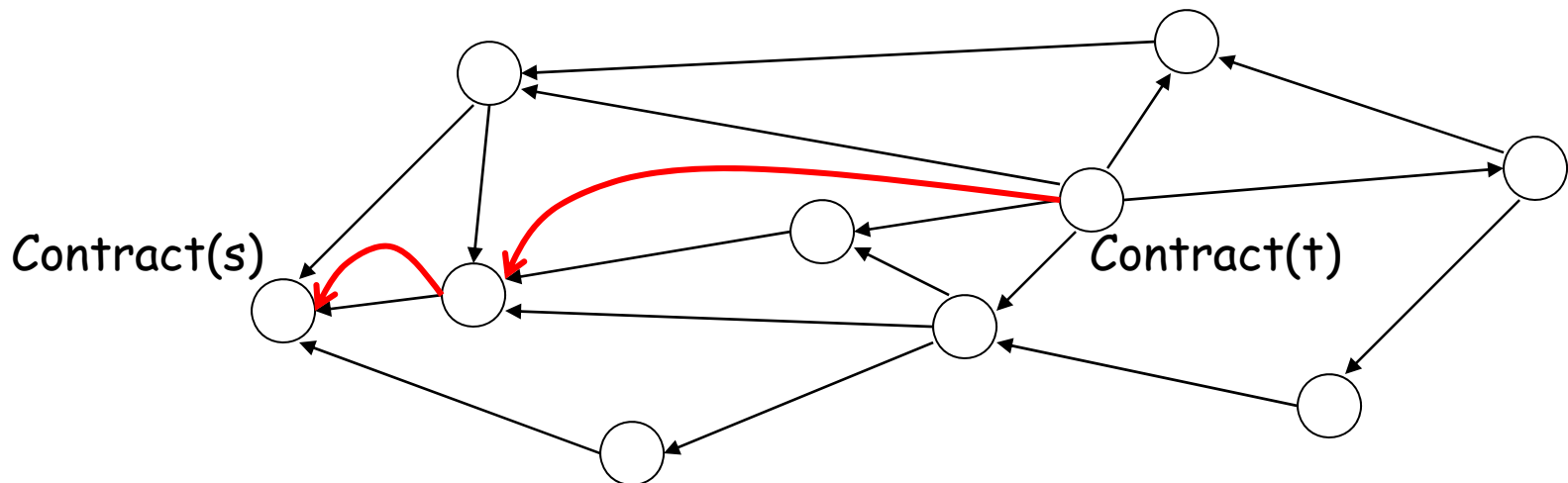
- find  $\text{contract}(t)$ - $\text{contract}(s)$  path, duplicate edges, “merge” edges within the same face
- construct the dual, except no edges cross the new path
- sum # paths between faces sharing a new edge



# “Forward-cuts” and different faces

What if  $s, t$  are on different faces ? [\[this paper\]](#)

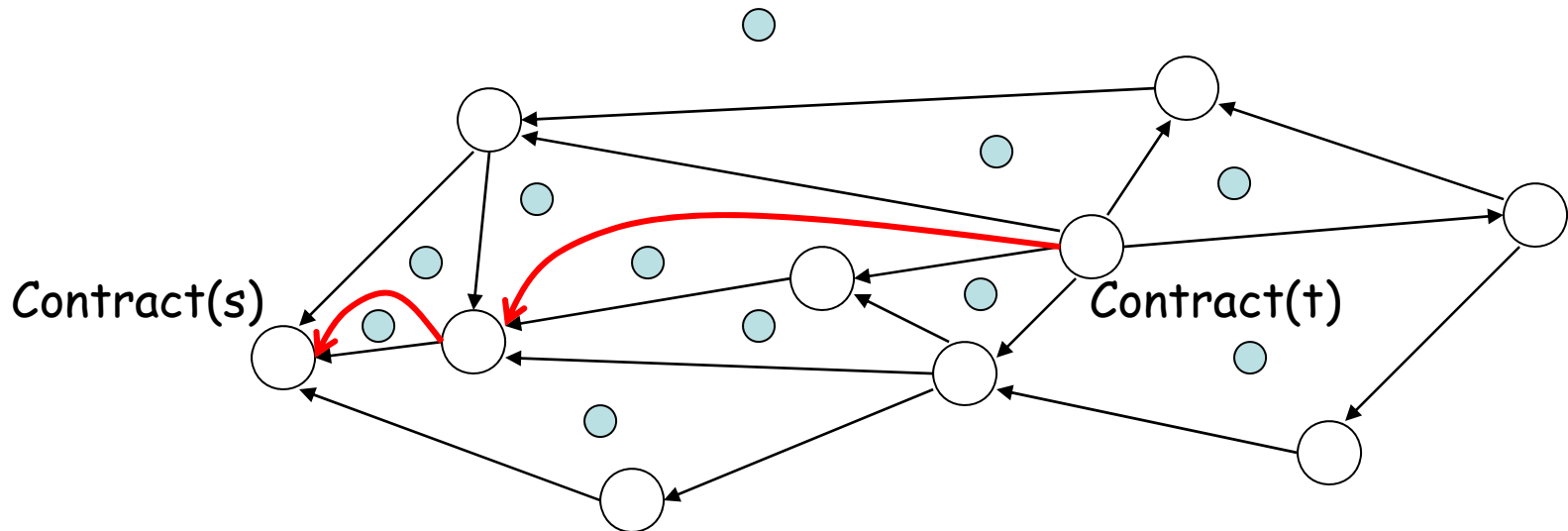
- find  $\text{contract}(t)$ - $\text{contract}(s)$  path, duplicate edges, “merge” edges within the same face
- construct the dual, except no edges cross the new path
- sum # paths between faces sharing a new edge



# “Forward-cuts” and different faces

What if  $s, t$  are on different faces ? [\[this paper\]](#)

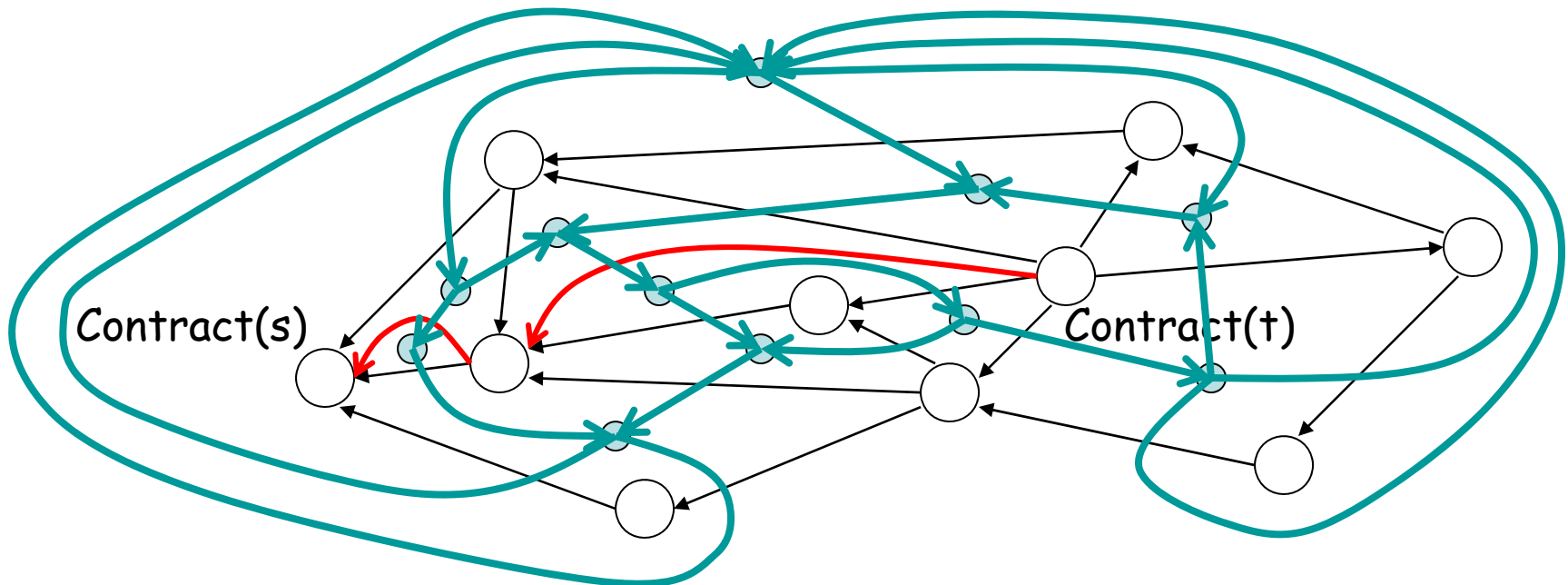
- find  $\text{contract}(t)$ - $\text{contract}(s)$  path, duplicate edges, “merge” edges within the same face
- construct the dual, except no edges cross the new path
- sum # paths between faces sharing a new edge



# “Forward-cuts” and different faces

What if  $s, t$  are on different faces? [[this paper](#)]

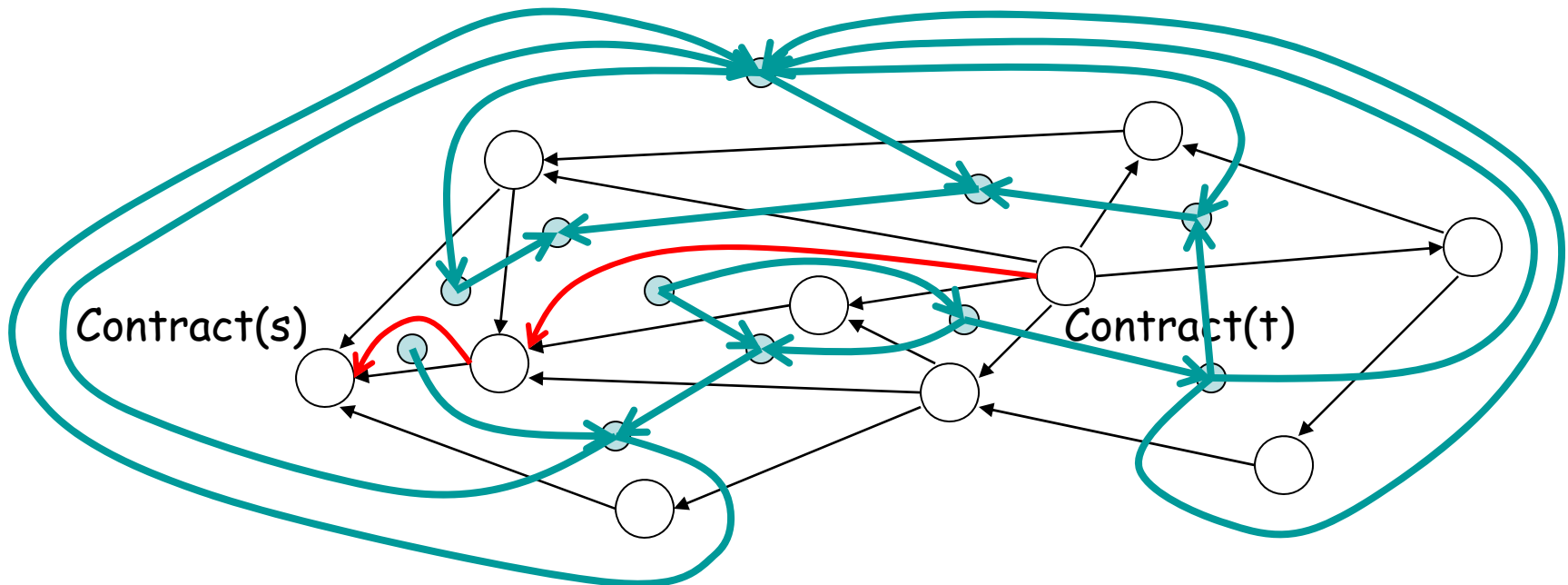
- find  $\text{contract}(t)$ - $\text{contract}(s)$  path, duplicate edges, “merge” edges within the same face
- construct the dual, except no edges cross the new path
- sum # paths between faces sharing a new edge



# “Forward-cuts” and different faces

What if  $s, t$  are on different faces? [[this paper](#)]

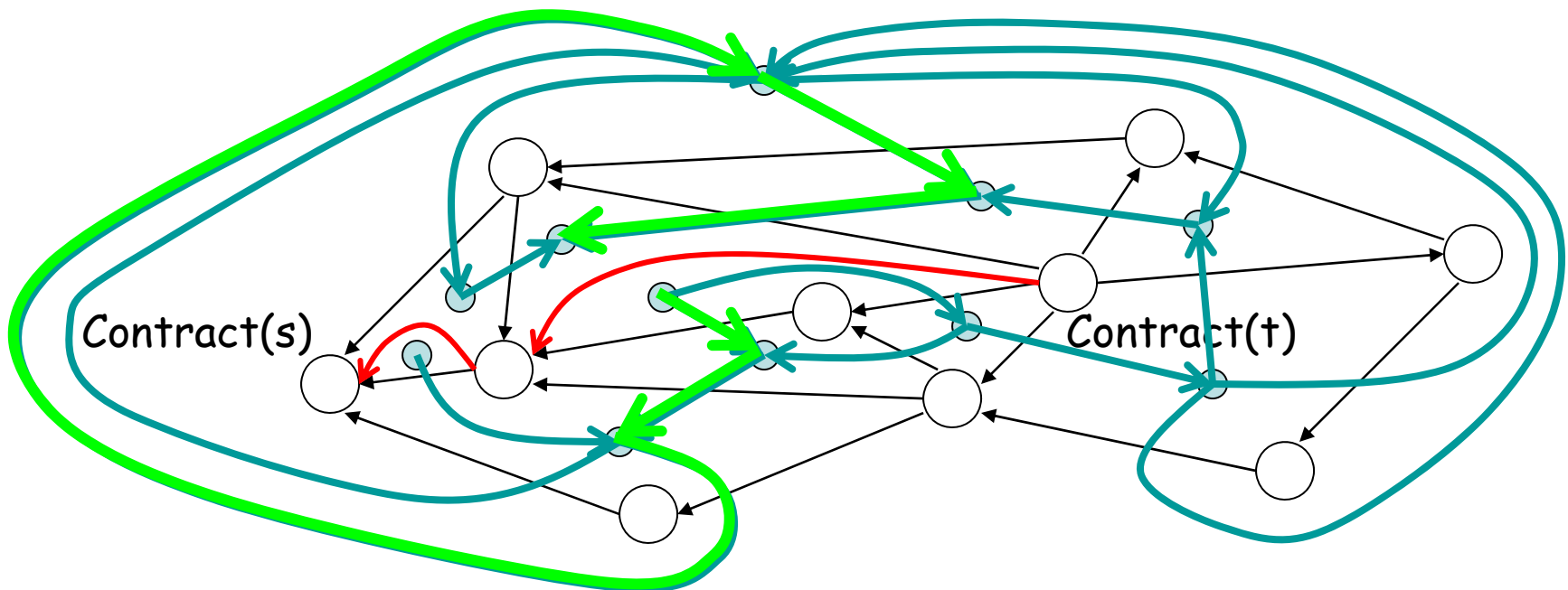
- find  $\text{contract}(t)$ - $\text{contract}(s)$  path, duplicate edges, “merge” edges within the same face
- construct the dual, except no edges cross the new path  $\rightarrow$  a **DAG**
- sum # paths between faces sharing a new edge



# “Forward-cuts” and different faces

What if  $s, t$  are on different faces ? [\[this paper\]](#)

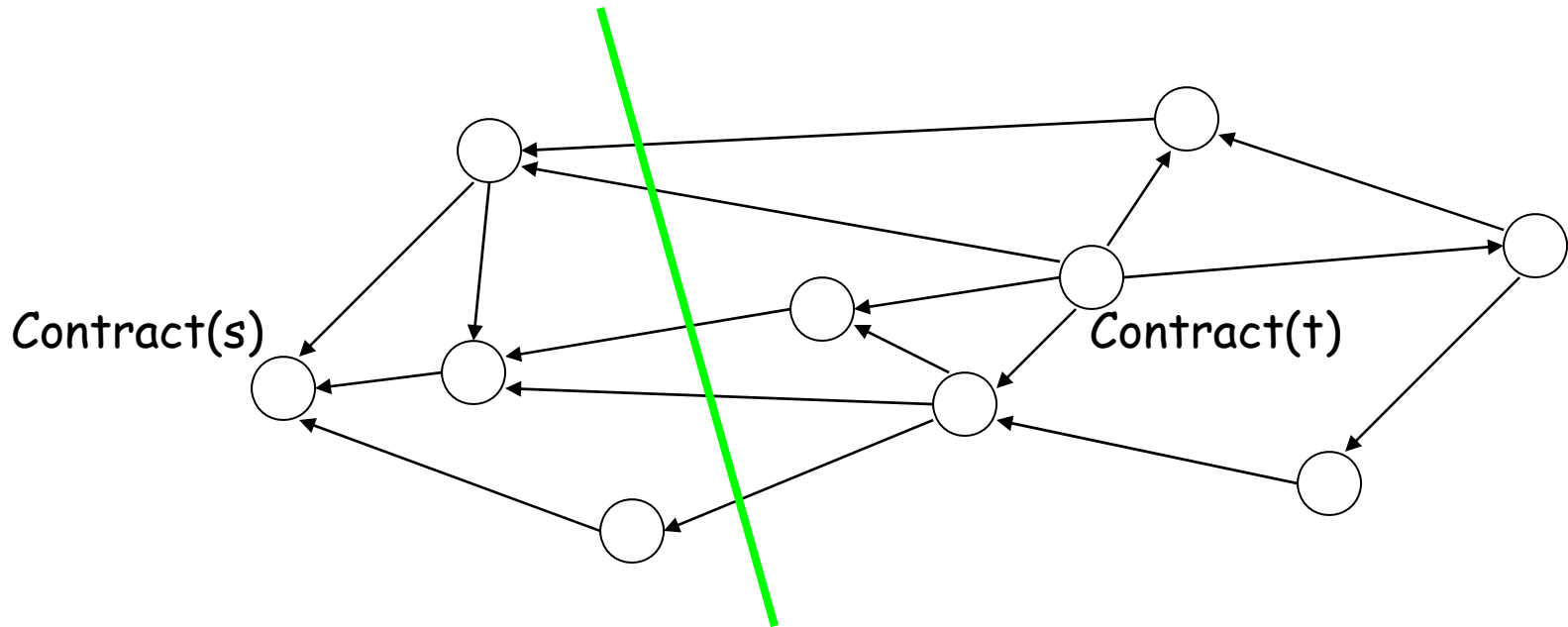
- find  $\text{contract}(t)$ - $\text{contract}(s)$  path, duplicate edges, “merge” edges within the same face
- construct the dual, except no edges cross the new path
- sum # paths between faces sharing a new edge



# “Forward-cuts” and different faces

What if  $s, t$  are on different faces ? [\[this paper\]](#)

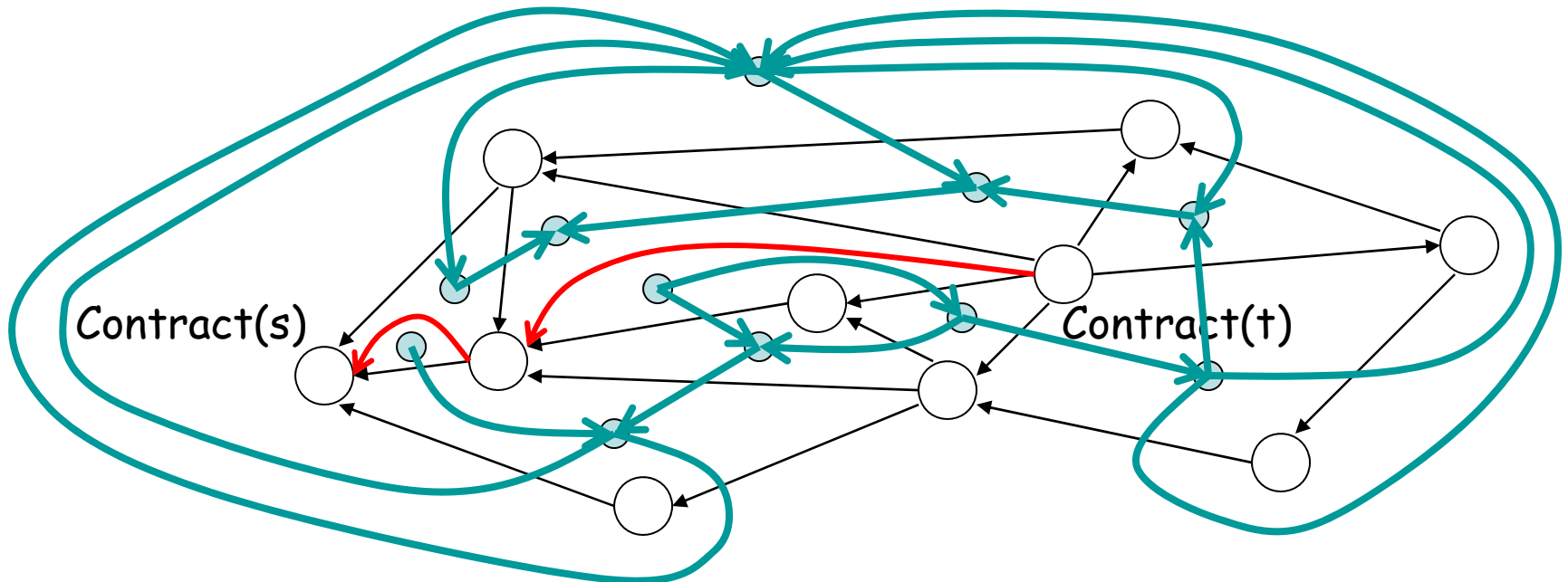
- find  $\text{contract}(t)$ - $\text{contract}(s)$  path, duplicate edges, “merge” edges within the same face
- construct the dual, except no edges cross the new path
- sum # paths between faces sharing a new edge



# “Forward-cuts” and different faces

Why does it work ?

- the dual (without the cross edges) is a DAG [whenever all vertices lie on an s-t path - a standard assumption]
- a simple dynamic programming counts all paths between two end-points in a DAG



# Running time

## Reduction to forward cuts:

- $O(n \log n)$  to find a (acyclic) max-flow in planar graphs  
[Borradaile-Klein '09]
- $O(n)$  to find and contract the strongly connected components

## Counting forward cuts:

- $O(n)$  find the path, construct the dual graph
- $O(n)$  compute #paths between two end-points in the dual
- $O(dn)$  overall computation of paths, at most  $d$  end-point pairs where  $d$  = length of the  $s$ - $t$  path

TOTAL:  $O(dn + n \log n)$

# Open Problems

Many open problems:

- non-planar graphs ? (unweighted or weighted)
- graph arising in computer vision (high-dimensional grids, with extra source/sink vertices)
- computing the sum of all  $(s,t)$ -cut weights (i.e., weighted counting = the partition function) -> important for parameter estimation, e.g., in the Markov Random Field model in vision

