Announcements

- Homework 0 Solutions online
- Homework 1 online due Friday
- Remember FinAid survey Due Friday
- Minor office hour schedule changes *this week*
  - Akhila: Thursday 4-6pm -> Friday 7-9pm
  - Oishi: Tuesday 10-11am -> Thursday 3:30-4:30pm

Note on HW0 Q2

"Exponential" runtime means 2^{cn} or maybe $2^{n\text{^}c}$, but not $2^{f(n)}$ for any function f.

For example, $n = 2^{\log(n)}$ is not exponential and does not grow faster than polynomials.

$a(n) = 2^{\mathrm{sqrt}\{\log(n)\}}$ is definitely not exponential time.

Last Time

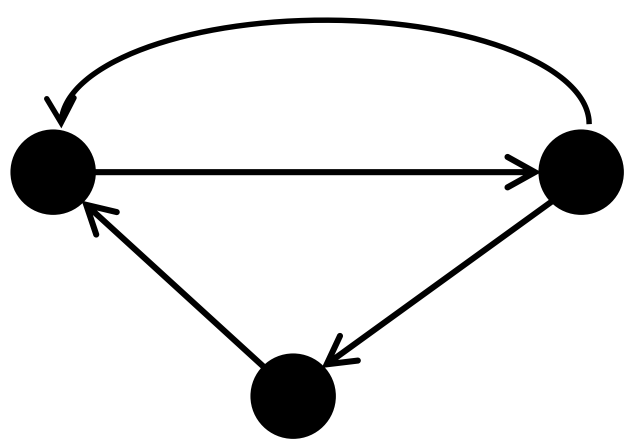
- Directed Graphs and dependency graphs
- Topological orderings
- DAGs

Directed Graphs

Often an edge makes sense both ways, but sometimes streets are one directional.

Definition: A directed graph is a graph where each edge has a direction. Goes *from* v *to* w.

Draw edges with arrows to denote direction.



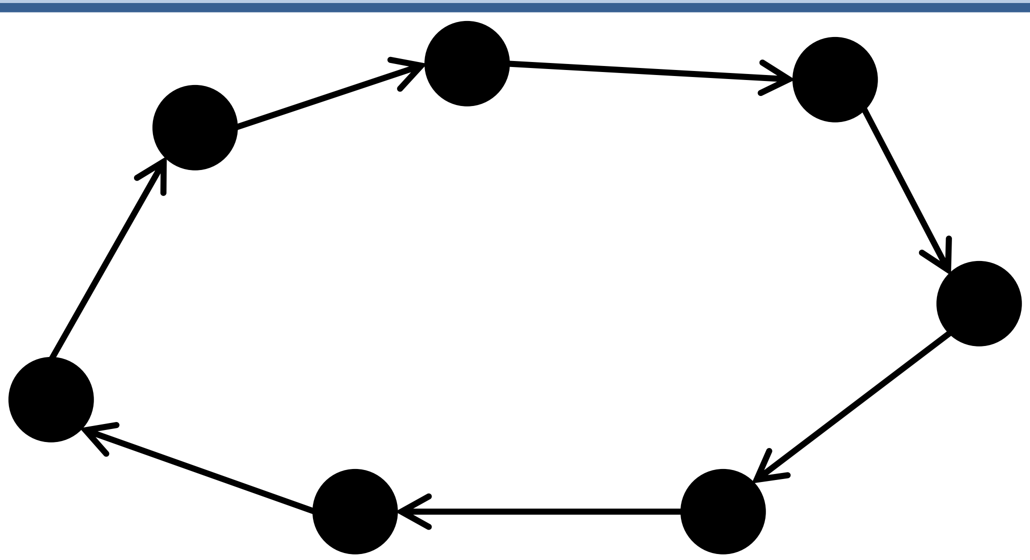
Dependency Graphs

A directed graph can be thought of as a graph of dependencies. Where an edge v→w means that v should come before w.

Definition: A topological ordering of a directed graph is an ordering of the vertices so that for each edge (v,w), v comes before w in the ordering.

Cycles

Definition: A cycle in a directed graph is a sequence of vertices v_1, v_2, v_3,…,v_n so that there are edges (v_1,v_2), (v_2,v_3),…,(v_{n-1},v_n),(v_n,v_1)



Obstacle

<u>Proposition:</u> If G is a directed graph with a cycle, then G has no topological ordering.

DAGs

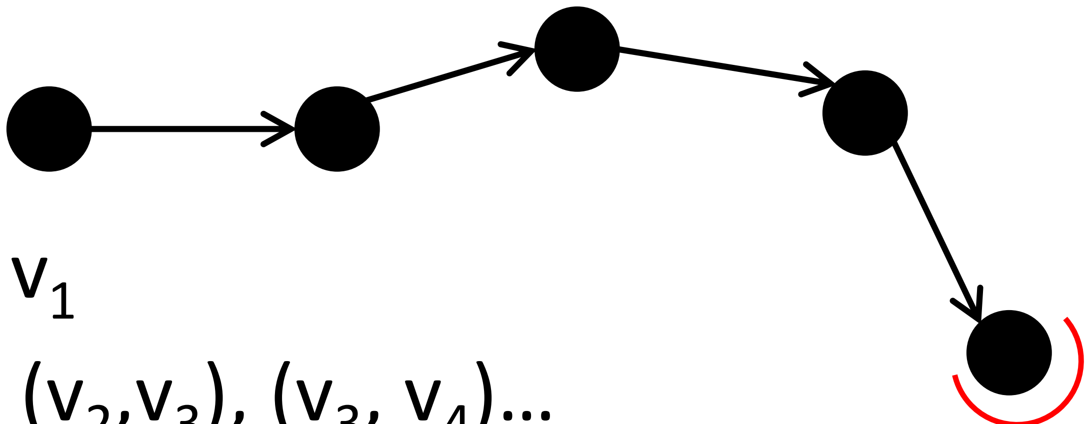
Definition: A Directed Acyclic Graph (DAG) is a directed graph which contains no cycles.

Existence of Orderings

<u>**Theorem:**</u> Let G be a (finite) DAG. Then G has a topological ordering.

Sinks

Lemma: Every finite DAG contains at least one sink.



Proof:

- Start at vertex $v = v_1$
- Find edges (v_1,v_2), (v_2,v_3), (v_3, v_4)…
- Eventually either:
  - Some vertex repeats (create cycle)
  - Get stuck (found a sink)

Proof of Theorem

- Induction on |G|.
- Find sink v.
- Let G’ = G-v.
- Inductively order G’ (still a DAG).
- Add v to the end of the ordering.

Today

- Topological sort
- Strongly connected components
- Meta-graphs

Algorithm

Problem: Design an algorithm that given a DAG G computes a topological ordering on G.

Algorithm

<u>Problem:</u> Design an algorithm that given a DAG G computes a topological ordering on G.

- Find sink v

Algorithm

<u>Problem:</u> Design an algorithm that given a DAG G computes a topological ordering on G.

- Find sink v
  - Follow chain of vertices until stuck

Algorithm

<u>Problem:</u> Design an algorithm that given a DAG G computes a topological ordering on G.

- Find sink v
  - Follow chain of vertices until stuck
- Compute ordering on G-v

Algorithm

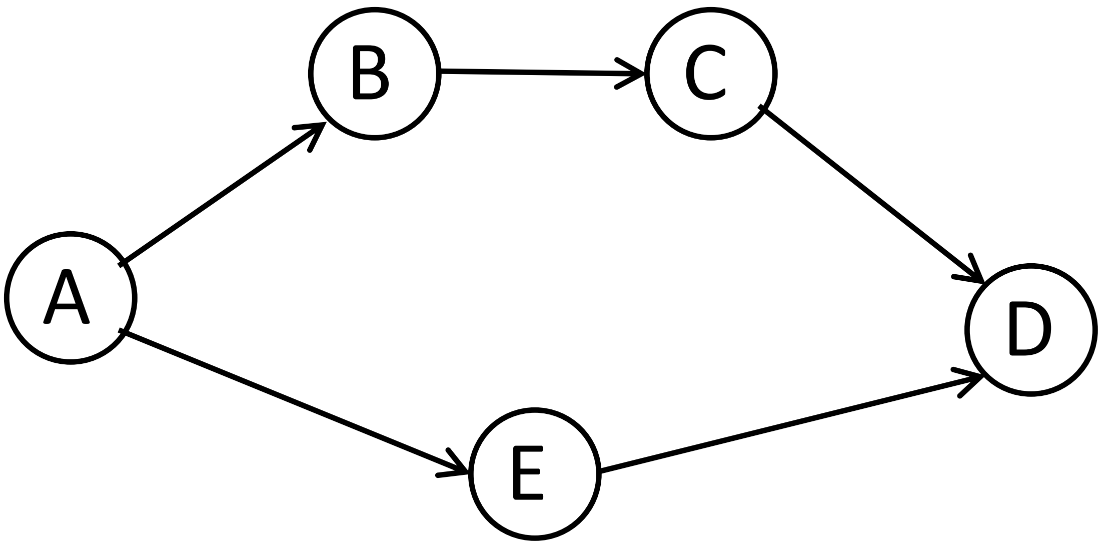
Problem: Design an algorithm that given a DAG G computes a topological ordering on G.

- Find sink v
  - Follow chain of vertices until stuck
- Compute ordering on G-v
- Place v at the end

Algorithm

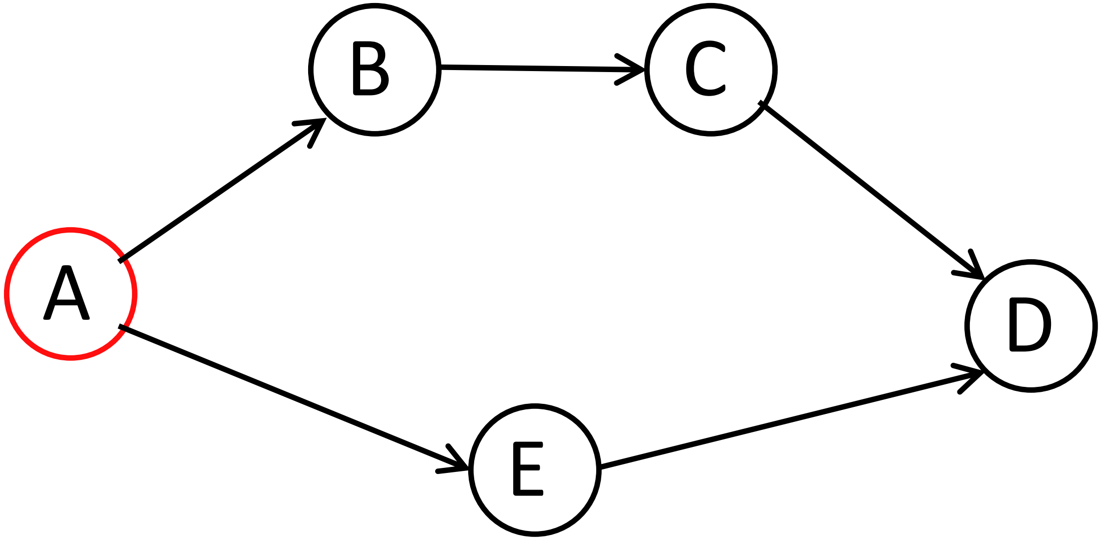
```
Ordering(G)
  If |G|=0, Return {}
  Let v ∈ G
  While there is an edge (v,w)
    v ← w
  Return (Ordering(G-v),v)
```

Example



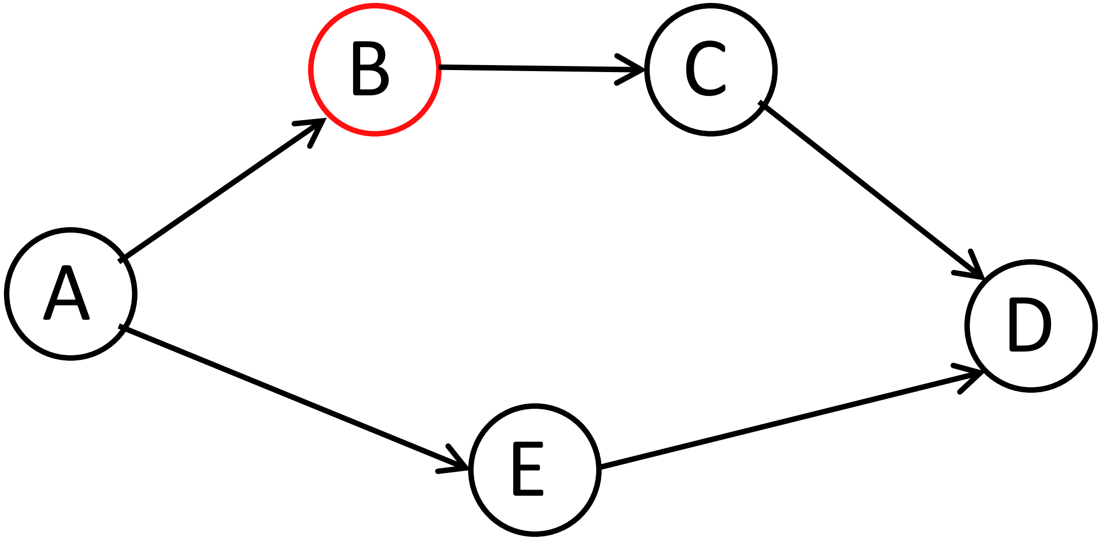
Final Ordering:

Example



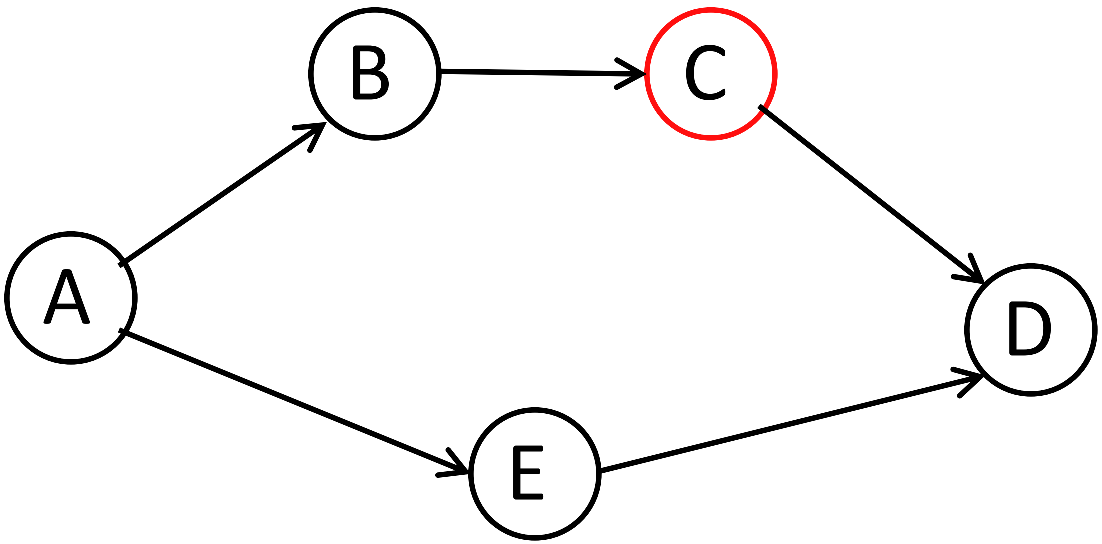
Final Ordering:

Example



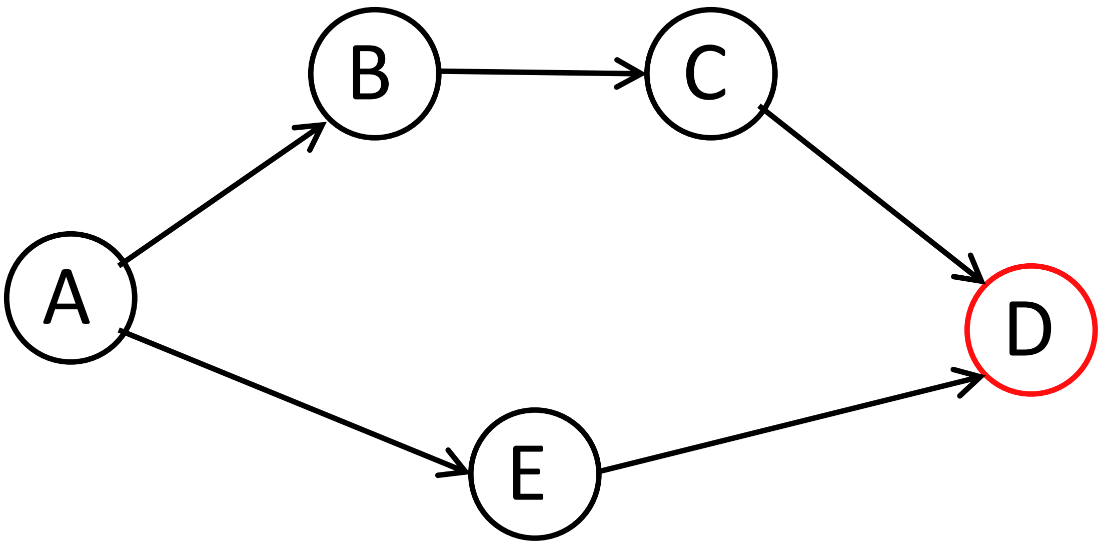
Final Ordering:

Example



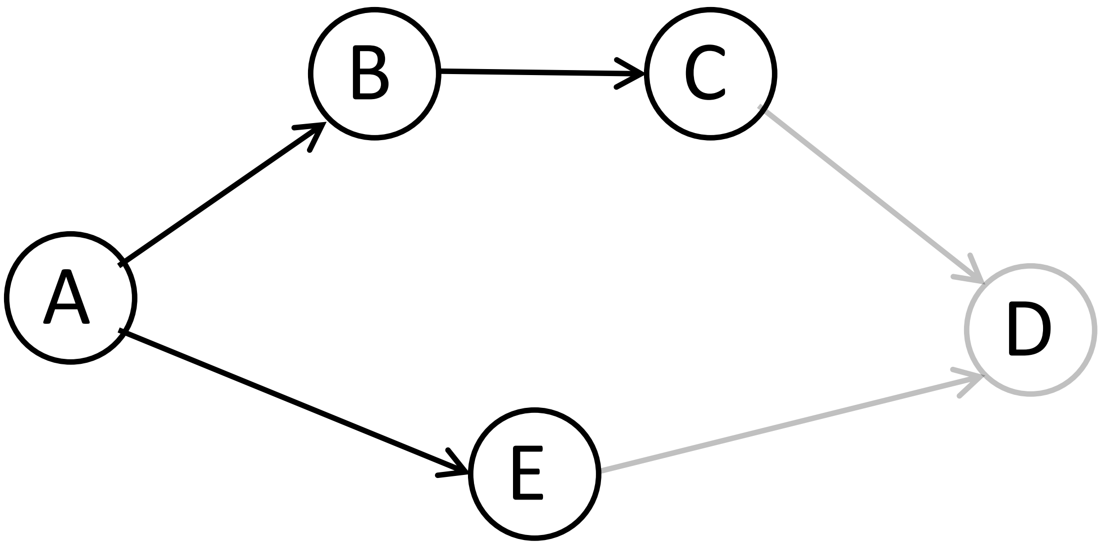
Final Ordering:

Example



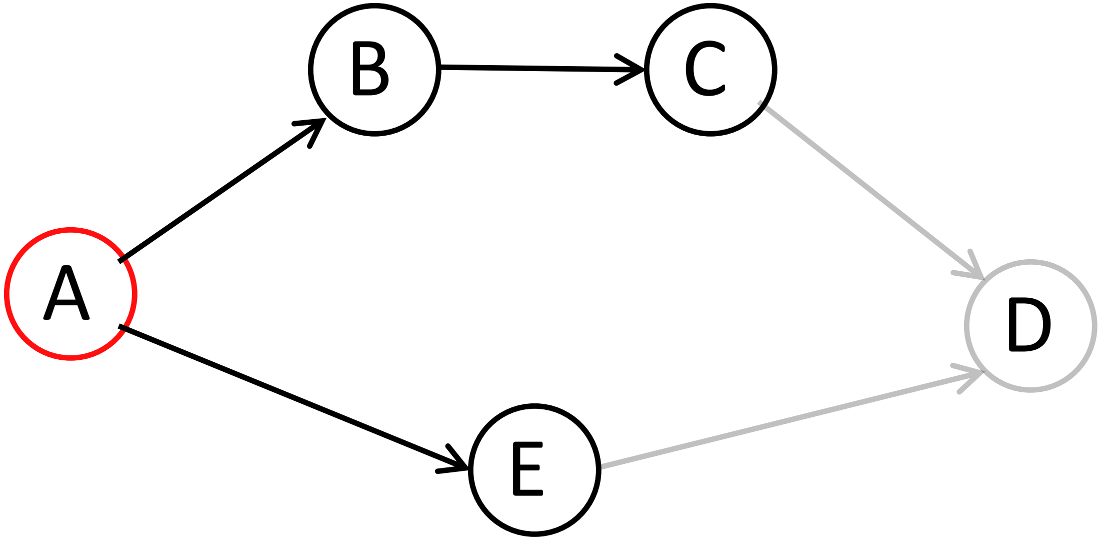
Final Ordering:

Example



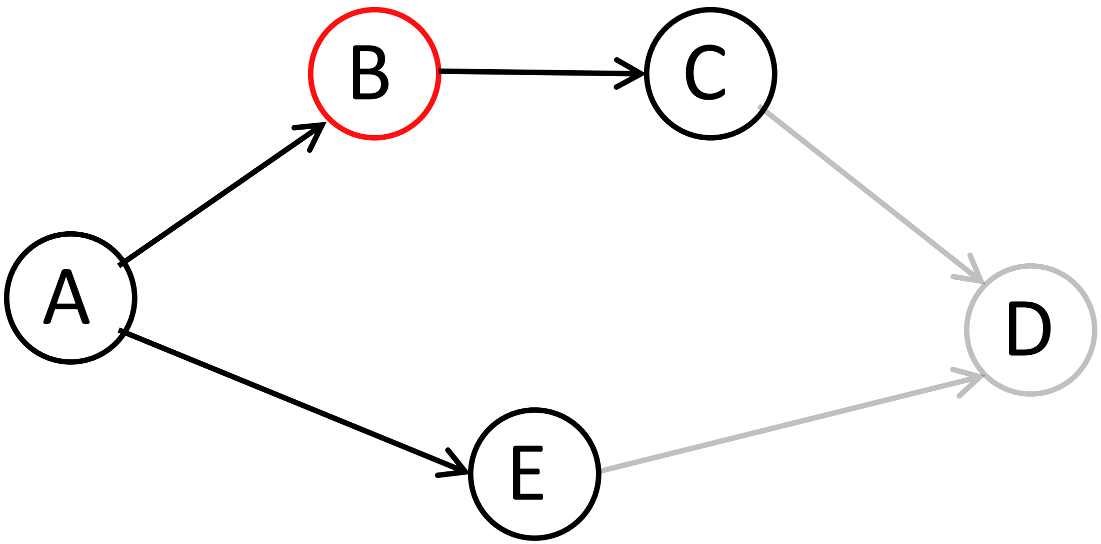
Final Ordering: D

Example



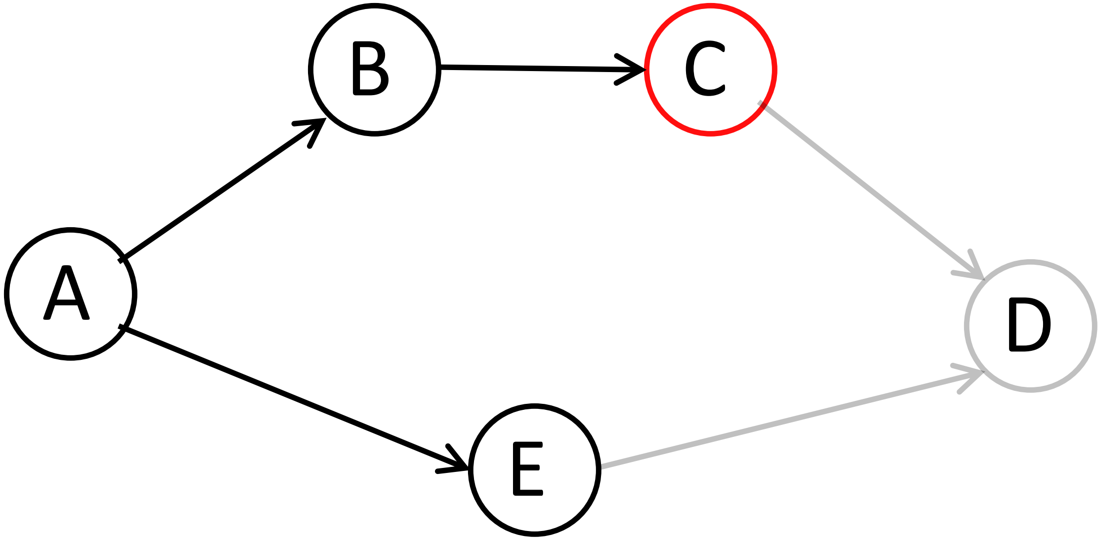
Final Ordering: D

Example



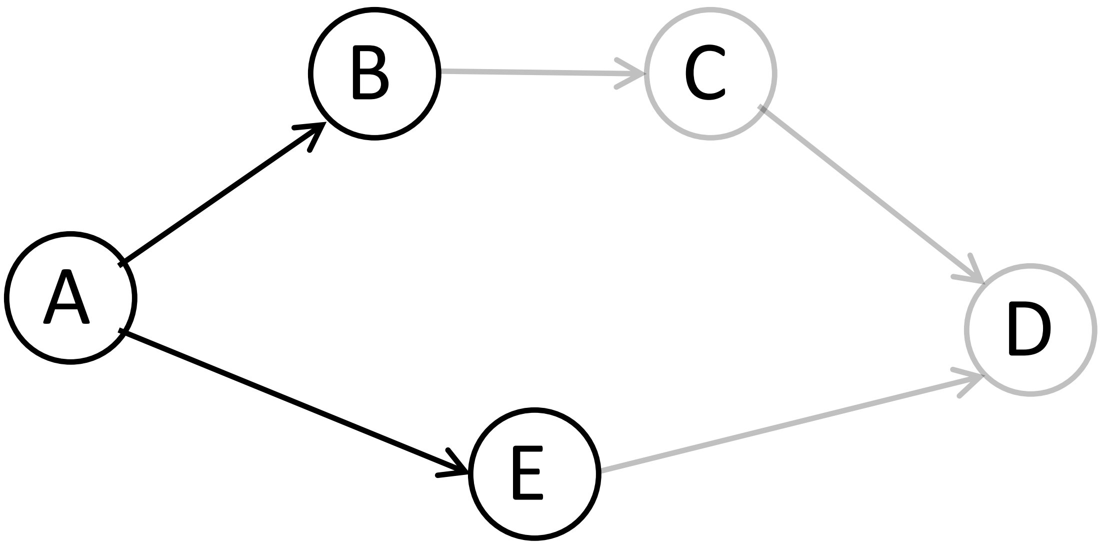
Final Ordering: D

Example



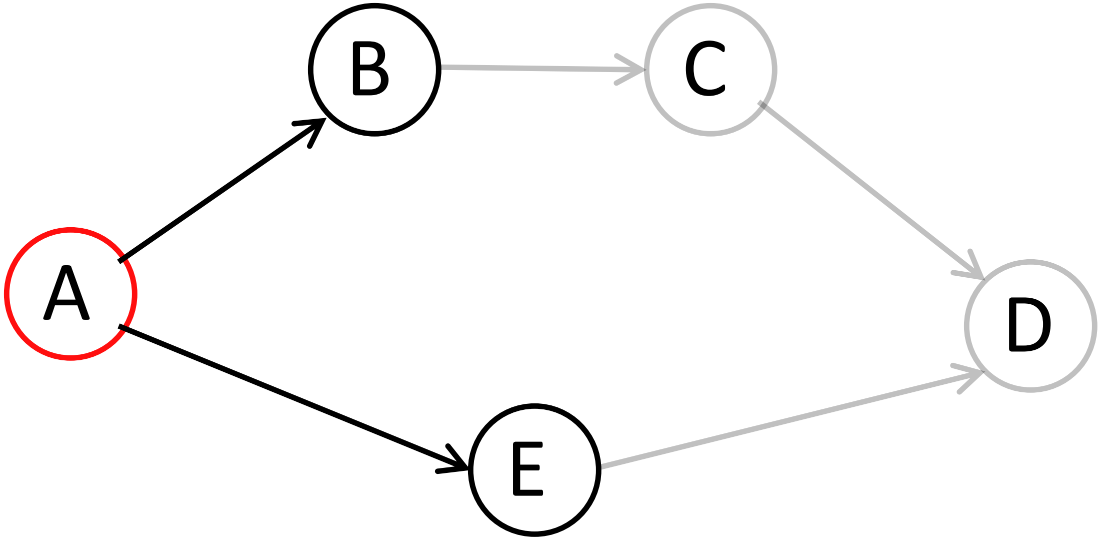
Final Ordering: D

Example



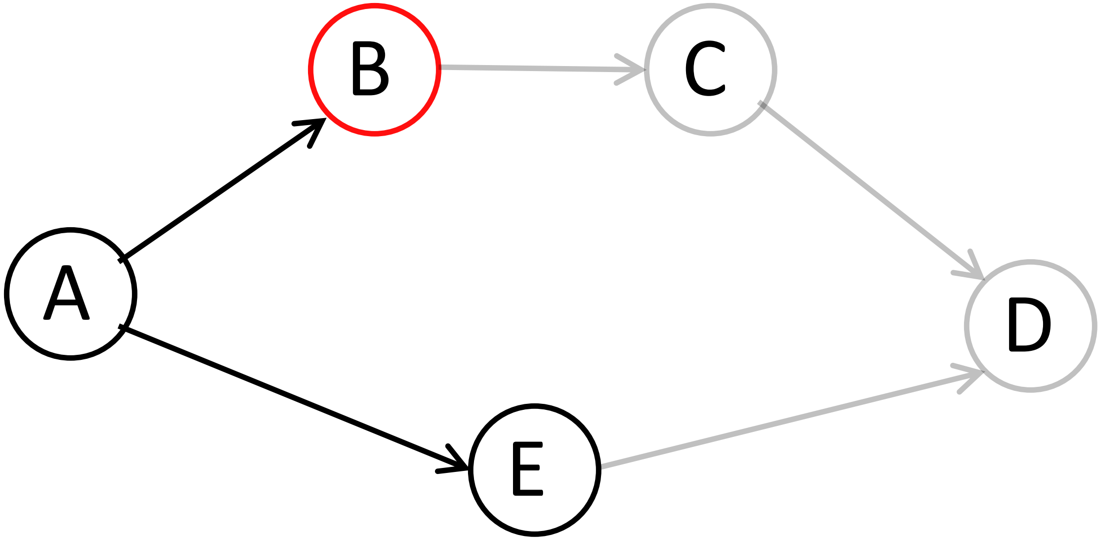
Final Ordering: C D

Example



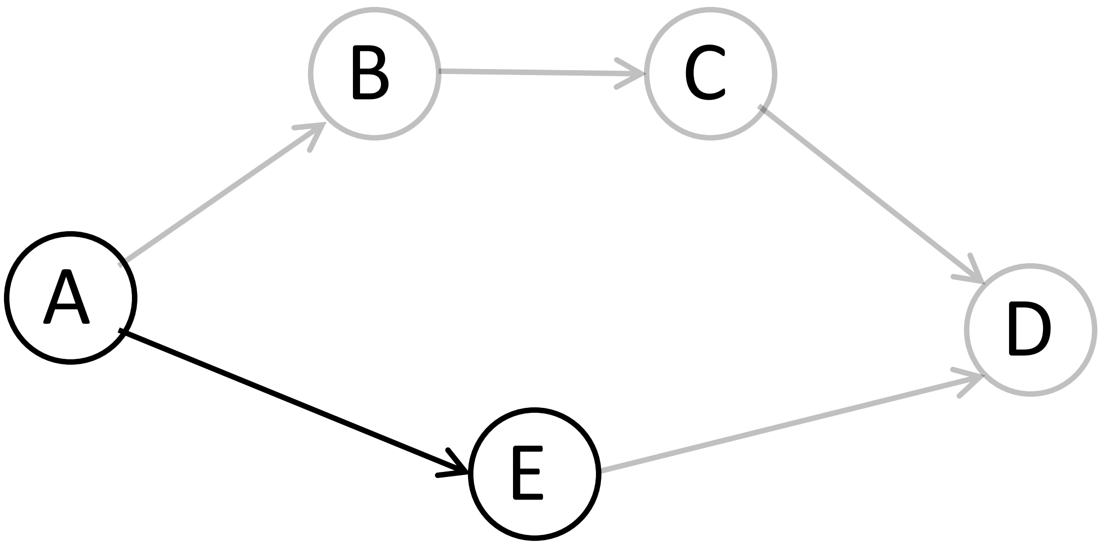
Final Ordering: C D

Example



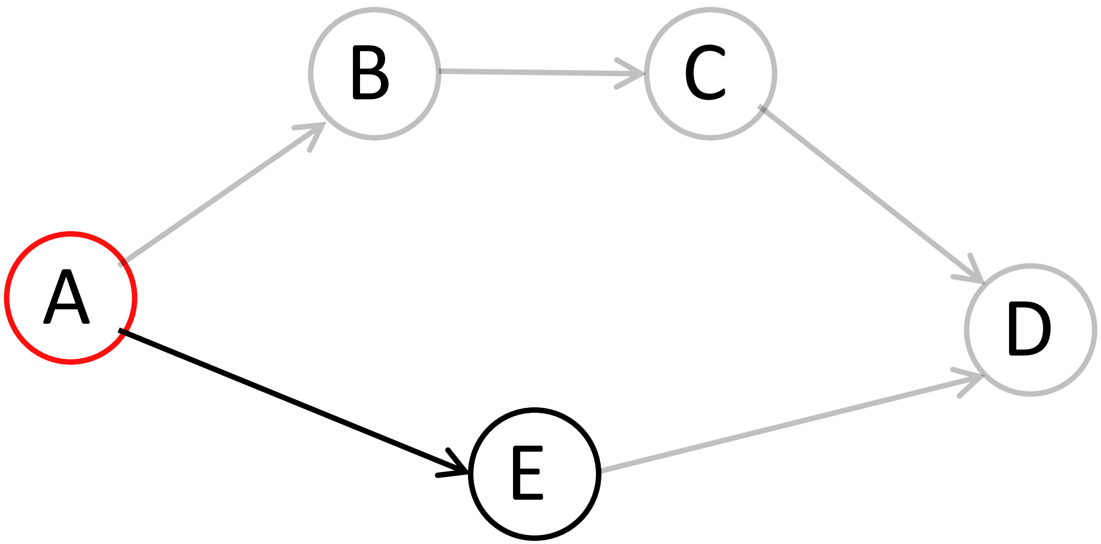
Final Ordering: C D

Example



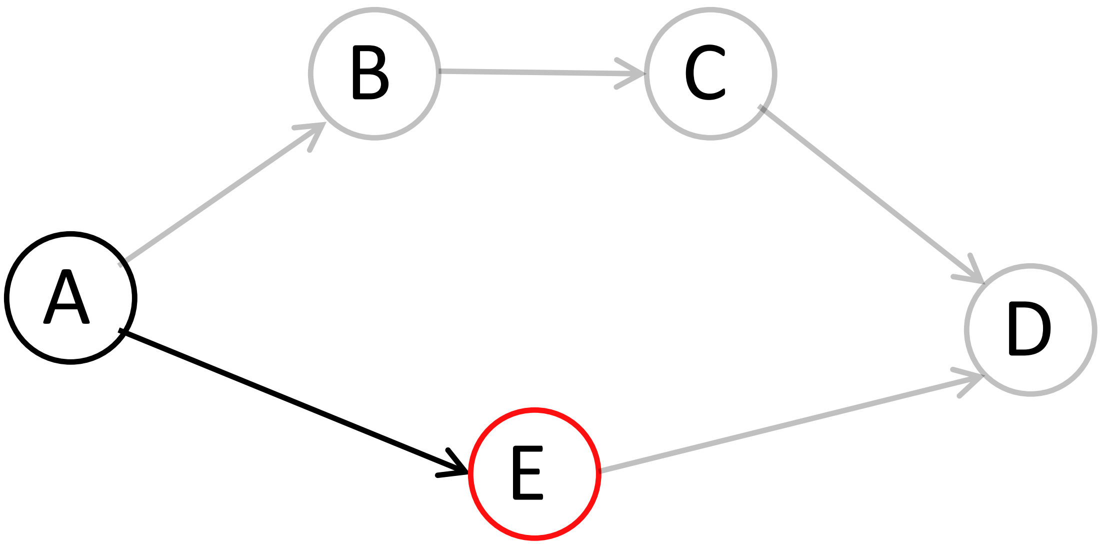
Final Ordering: B C D

Example



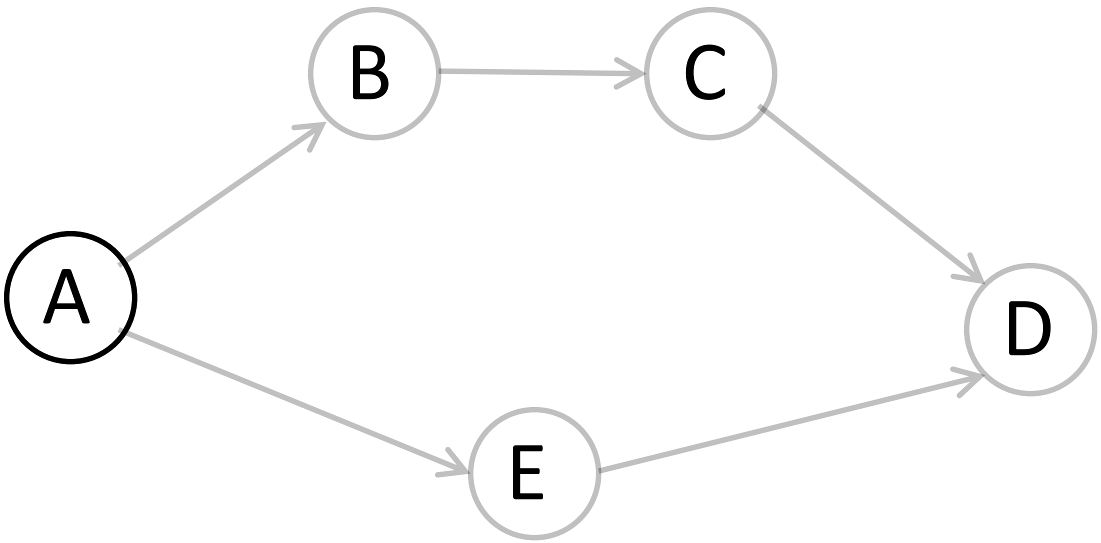
Final Ordering: B C D

Example



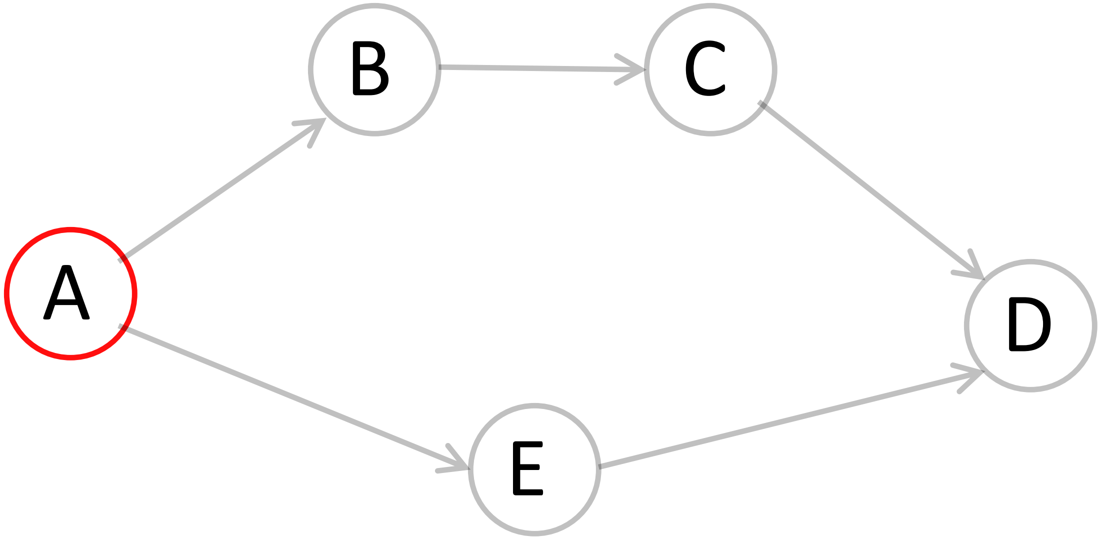
Final Ordering: B C D

Example



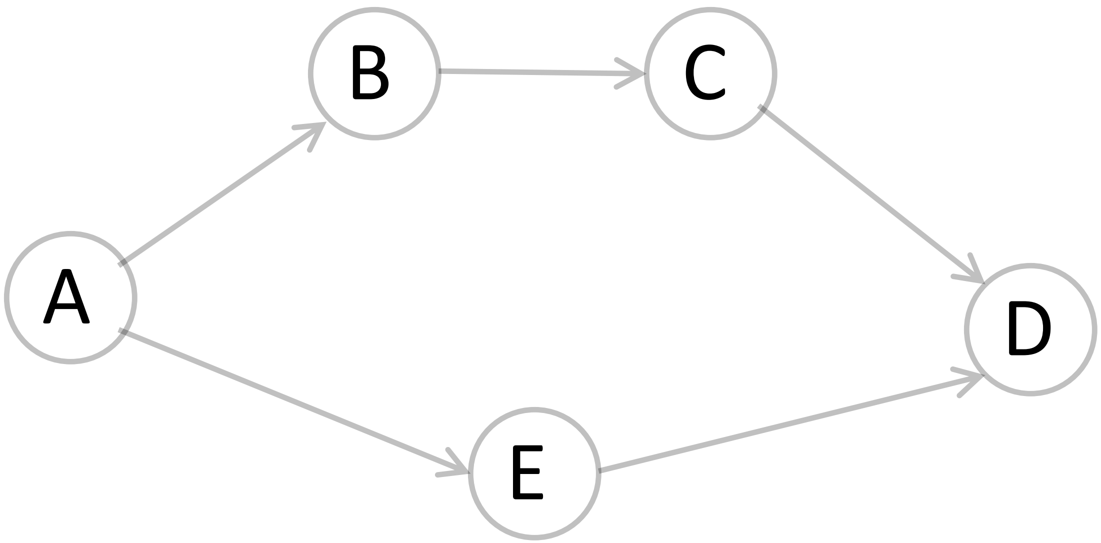
Final Ordering: E B C D

Example



Final Ordering: E B C D

Example



Final Ordering: A E B C D

Runtime

$(|V|$ time to find each sink$) \cdot (|V|$ sinks$)$
$= O(|V|^2)$ runtime.

Runtime

(|V| time to find each sink) ·(|V| sinks)
= $O(|V|^2)$ runtime.

This is suboptimal.

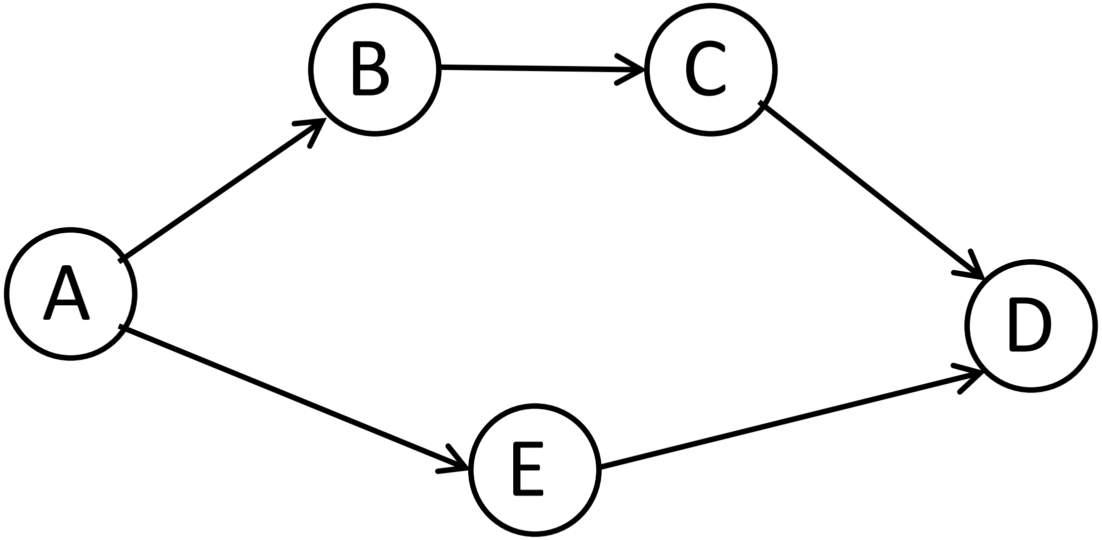
Runtime

(|V| time to find each sink) ·(|V| sinks)
= $O(|V|^2)$ runtime.

This is suboptimal.

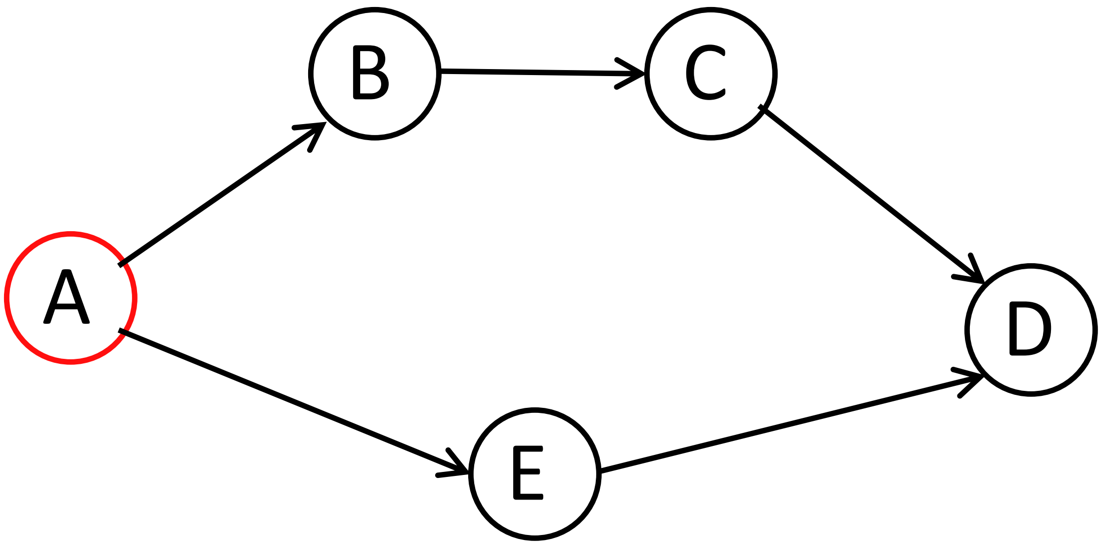
Problem: After adding a sink to the end, the algorithm forgets the path that it took. Instead of backing up to start, just back up one step.

Example II



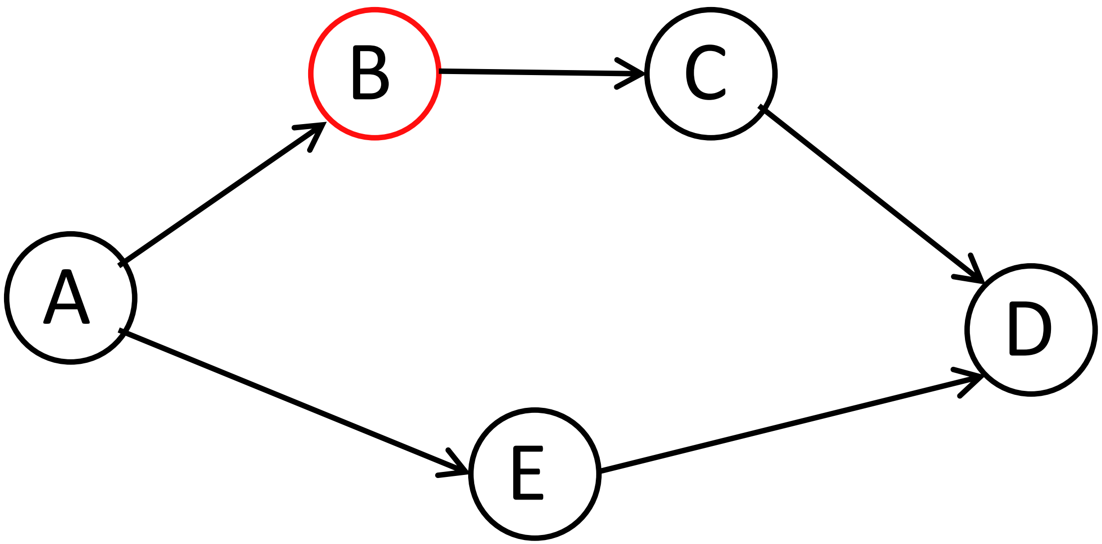
Final Ordering:

Example II



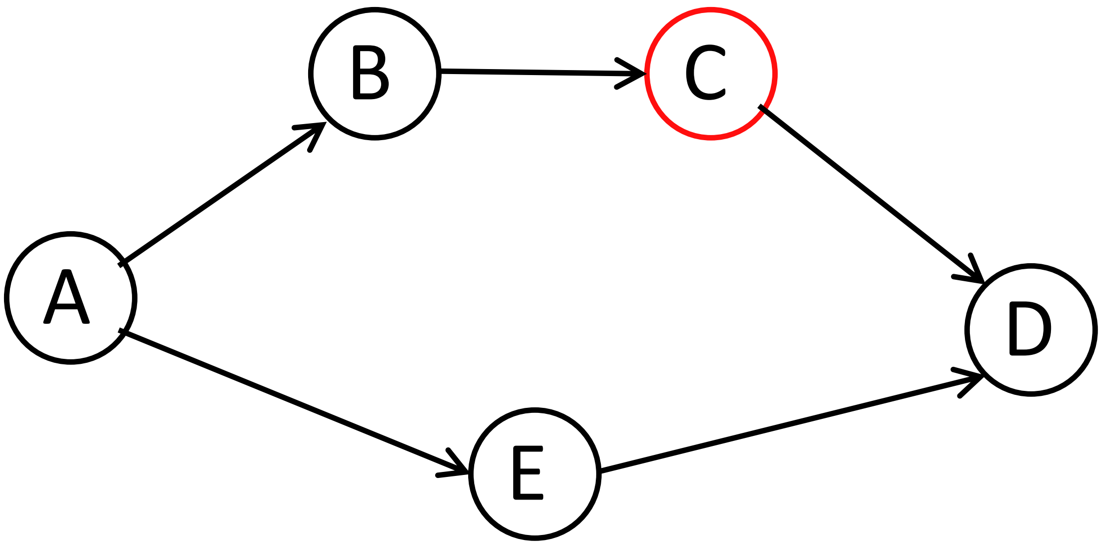
Final Ordering:

Example II



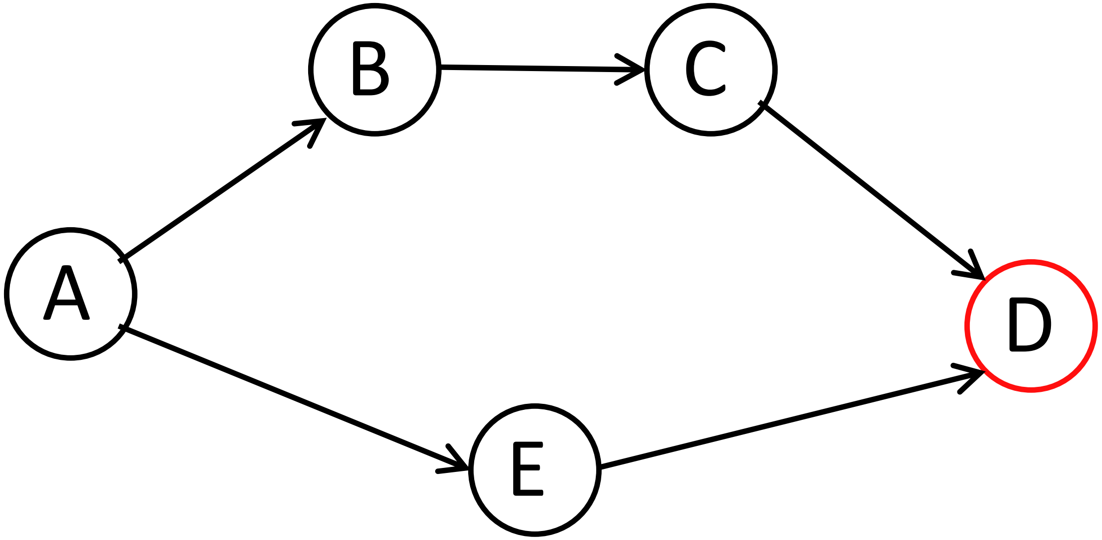
Final Ordering:

Example II



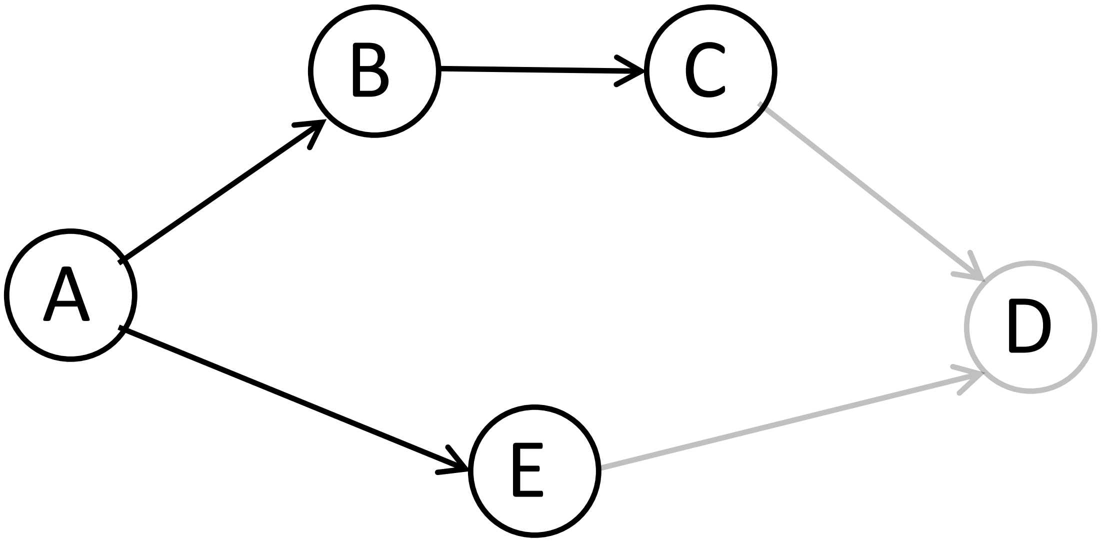
Final Ordering:

Example II



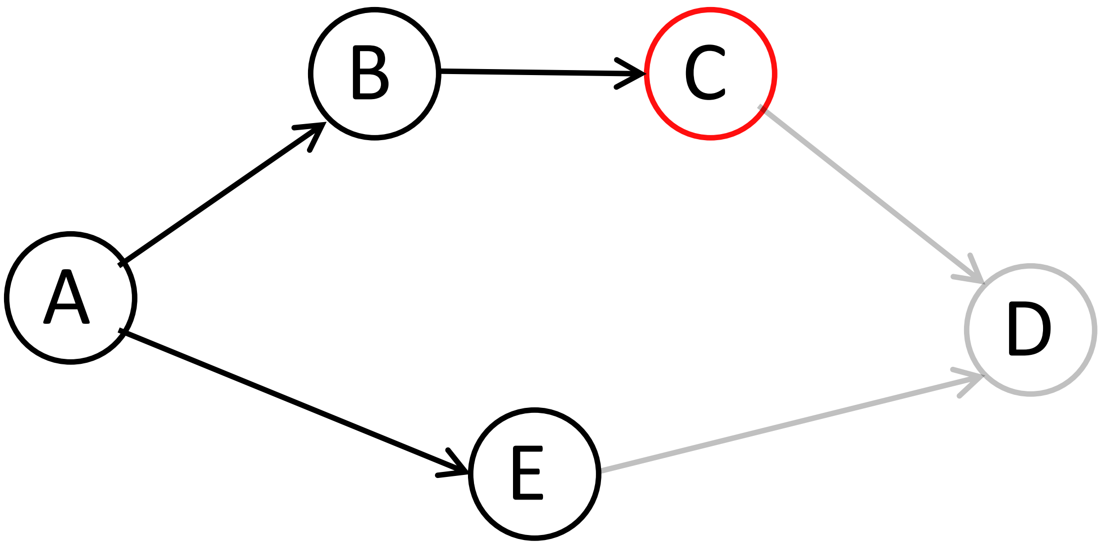
Final Ordering:

Example II



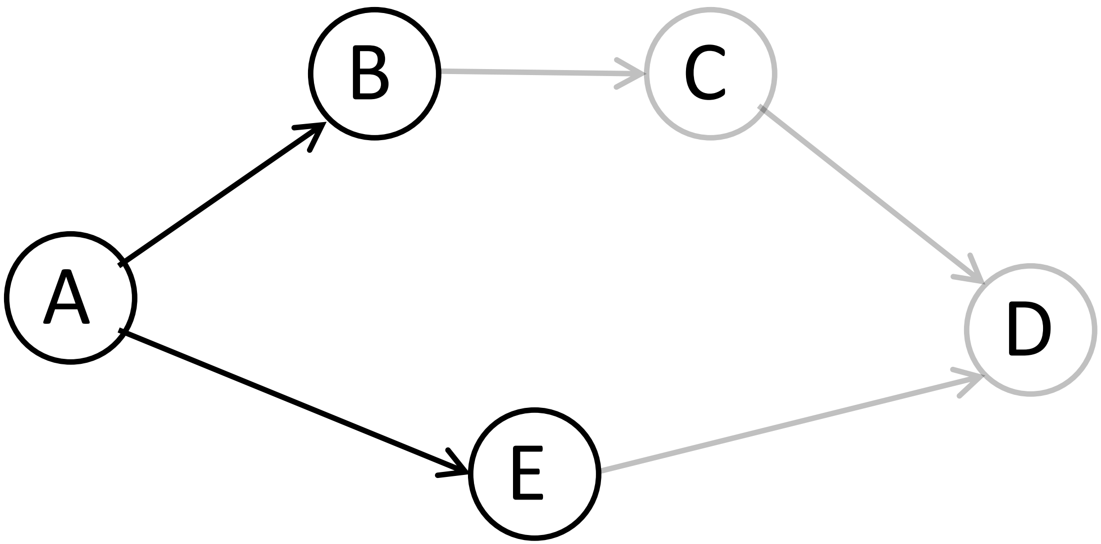
Final Ordering: D

Example II



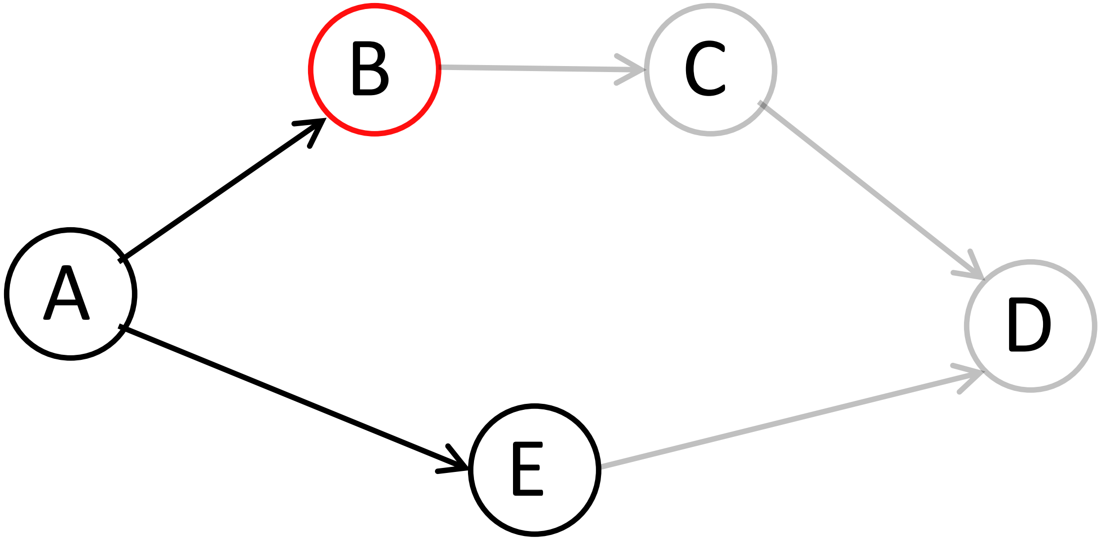
Final Ordering: D

Example II



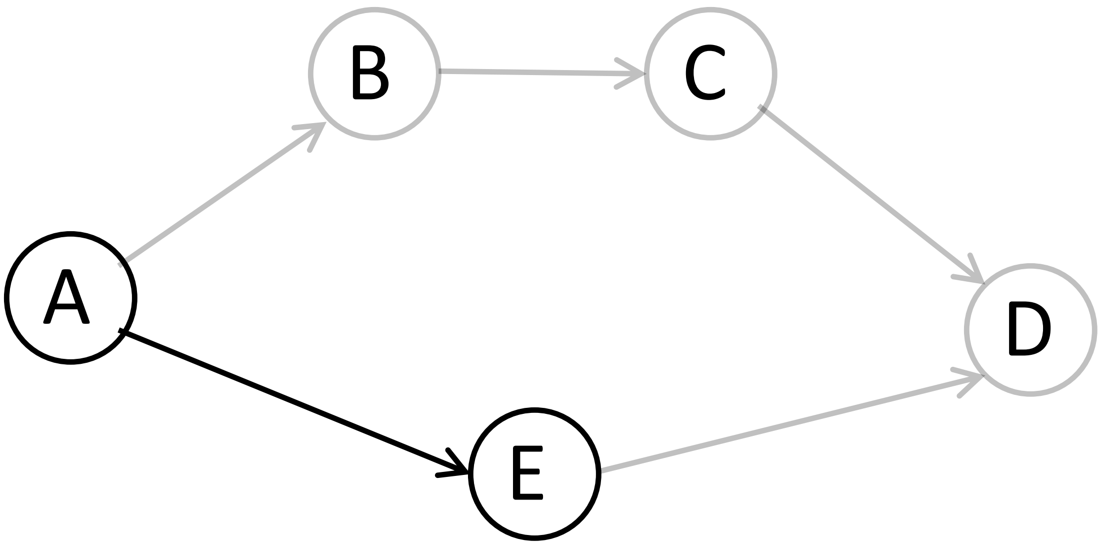
Final Ordering: C D

Example II



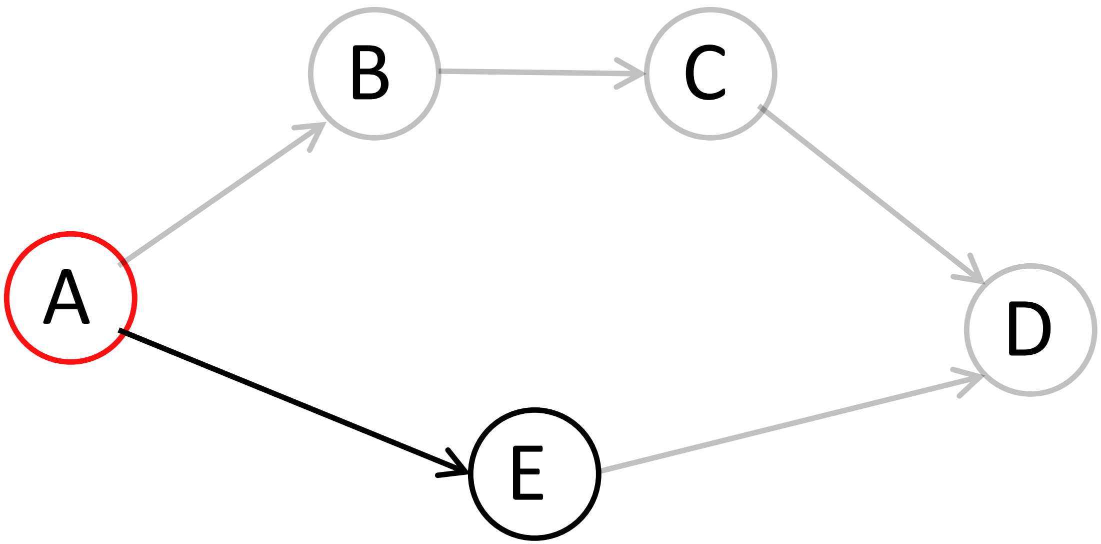
Final Ordering: C D

Example II



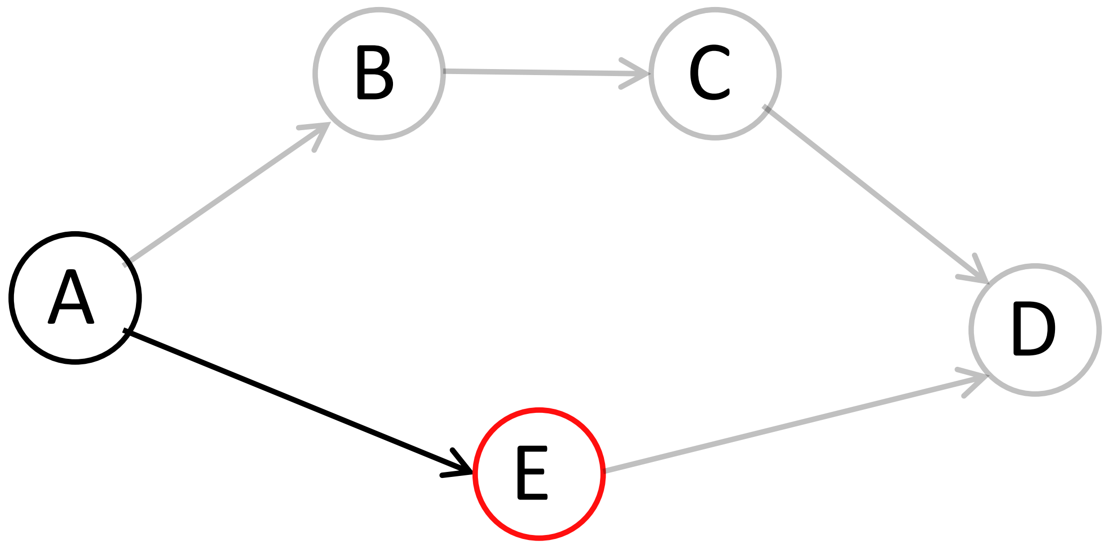
Final Ordering: B C D

Example II



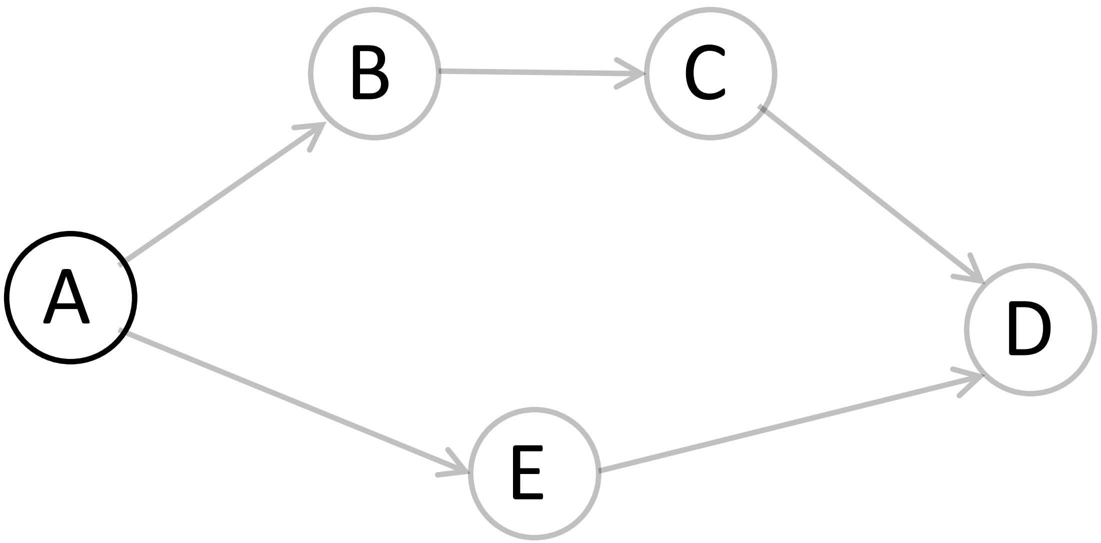
Final Ordering: B C D

Example II



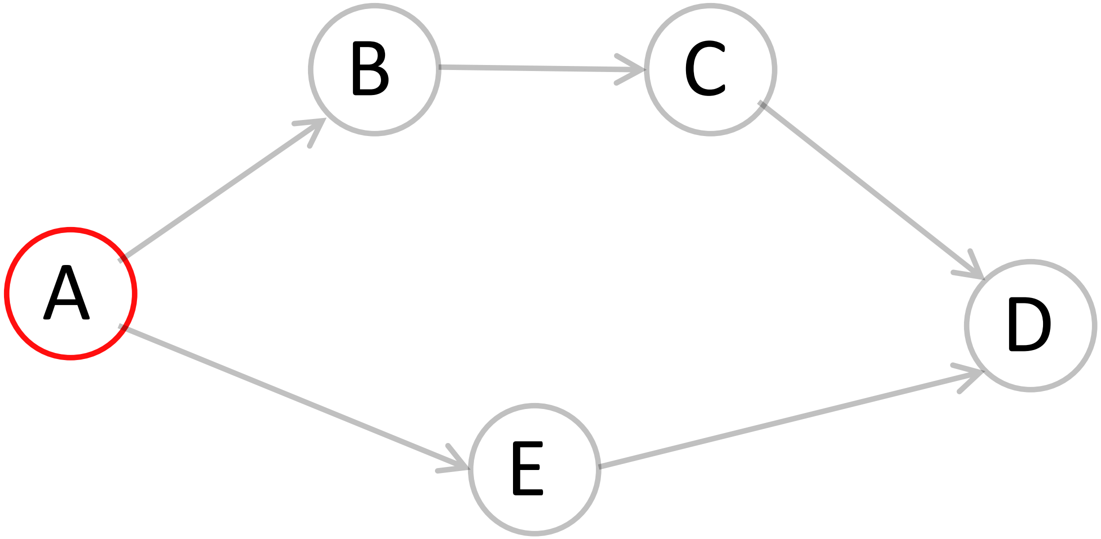
Final Ordering: B C D

Example II



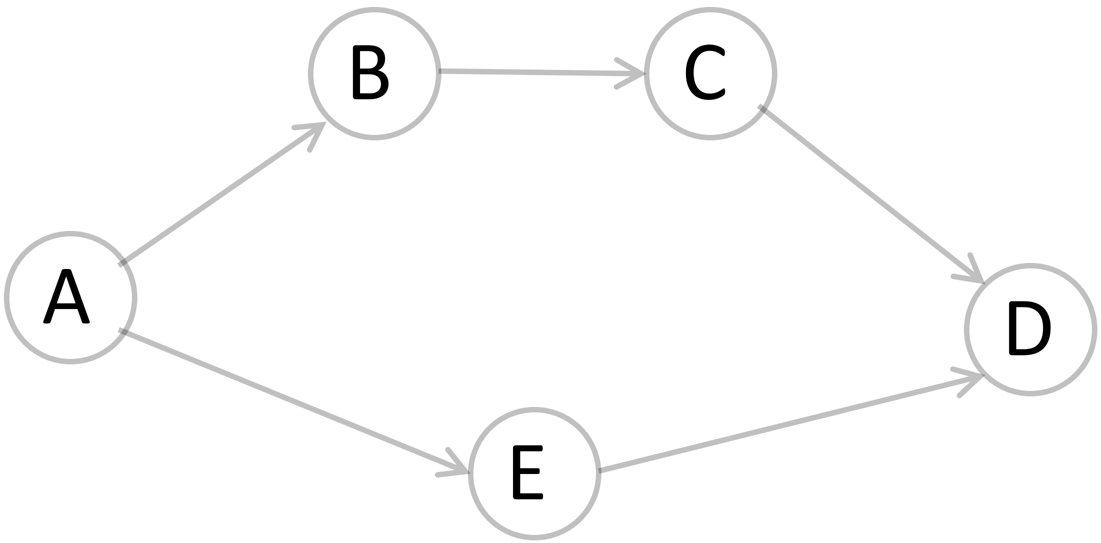
Final Ordering: E B C D

Example II



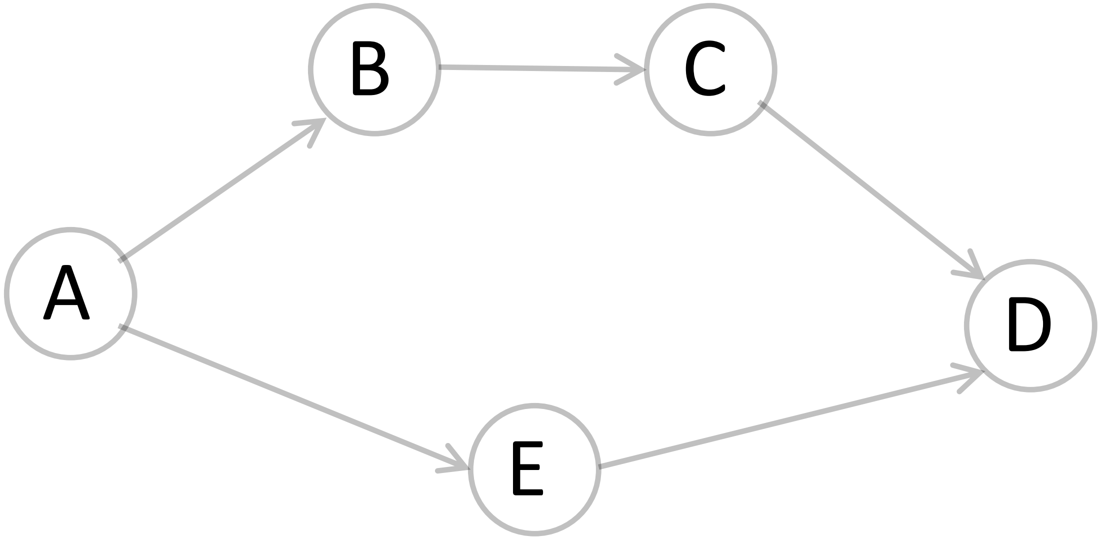
Final Ordering: E B C D

Example II



Final Ordering: A E B C D

Example II



This is just DFS ordering!

Final Ordering: A E B C D

Algorithm II

```
TopologicalSort(G)
   Run DFS(G) w/ pre/post numbers
   Return the vertices in reverse
  postorder
```

Algorithm II

```
TopologicalSort(G)
   Run DFS(G) w/ pre/post numbers
   Return the vertices in reverse
  postorder
```

Note: Can add vertices to list as postorder assigned.

Algorithm II

```
TopologicalSort(G)
  Run DFS(G) w/ pre/post numbers
  Return the vertices in reverse
 postorder
```

Note: Can add vertices to list as postorder assigned.

Runtime: $O(|V|+|E|)$.

Correctness

<u>Proposition:</u> If G is a DAG with an edge v → w then `w.post < v.post`.

Correctness

Proposition: If G is a DAG with an edge v → w then `w.post < v.post`.

Implies that ordering is correct.

Correctness

<u>Proposition:</u> If G is a DAG with an edge v → w then `w.post < v.post`.

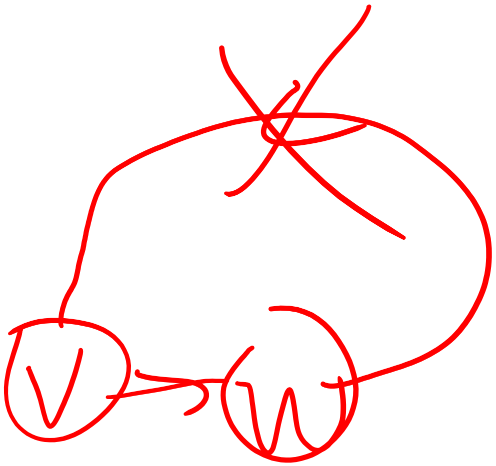
Implies that ordering is correct.

<u>Proof:</u>

Break into cases based on which of v or w is discovered first.

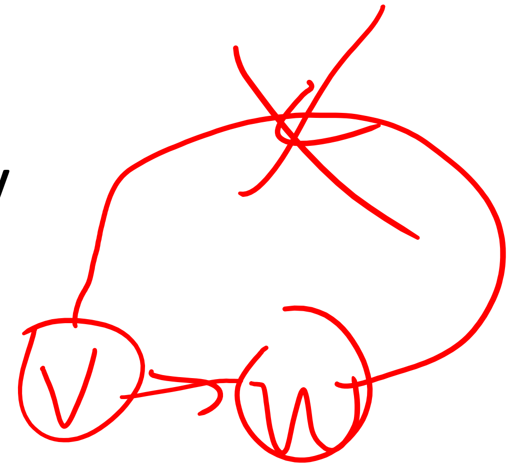
Proof

- If v discovered first
  - w discovered while exploring v
  - w descendant of v
  - pre-post intervals nested
  - `w.post < v.post`



Proof

- If v discovered first
  - w discovered while exploring v
  - w descendant of v
  - pre-post intervals nested
  - `w.post < v.post`
- If w discovered first
  - v cannot be a descendant (DAG)
  - pre-post intervals are disjoint
  - `w.post < v.post`



Topological Sort

Useful algorithm.

- Many graph algorithms are relatively easy to find the answer for v if you’ve already found the answer for everything downstream of v.

Topological Sort

Useful algorithm.

- Many graph algorithms are relatively easy to find the answer for v if you’ve already found the answer for everything downstream of v.
  - Topologically sort G.
  - Solve for v in reverse topological order.

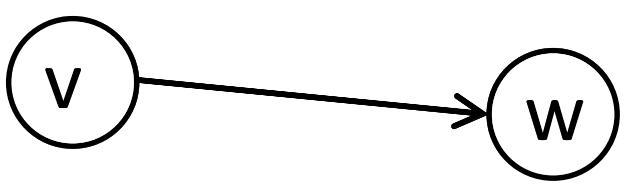
Connectivity in Digraphs

In undirected graphs, we had a very clean description of reachability: v was reachable from w if and only if they were in the same connected component.

Connectivity in Digraphs

In undirected graphs, we had a very clean description of reachability: v was reachable from w if and only if they were in the same connected component.

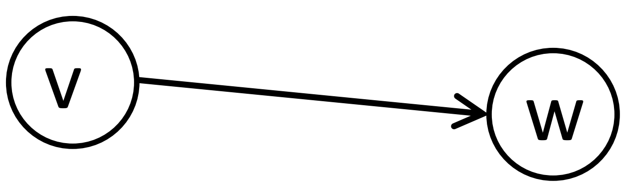
This no longer works for digraphs.



Connectivity in Digraphs

In undirected graphs, we had a very clean description of reachability: v was reachable from w if and only if they were in the same connected component.

This no longer works for digraphs.



What is the right notion of connected components for digraphs?

Problems

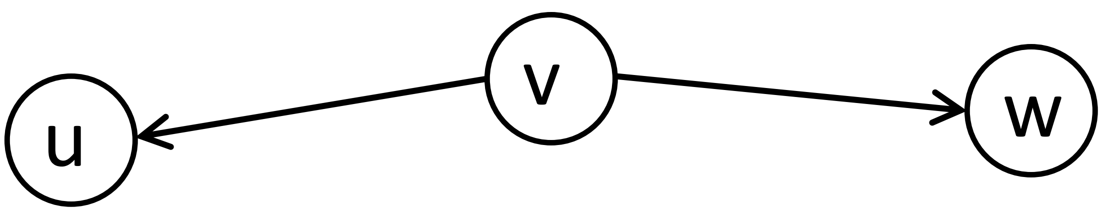
- Reachability no longer symmetric
  - can reach w from v but not v from w.

Problems

- Reachability no longer symmetric
  - can reach w from v but not v from w.
- Maybe allow reachability in either direction?

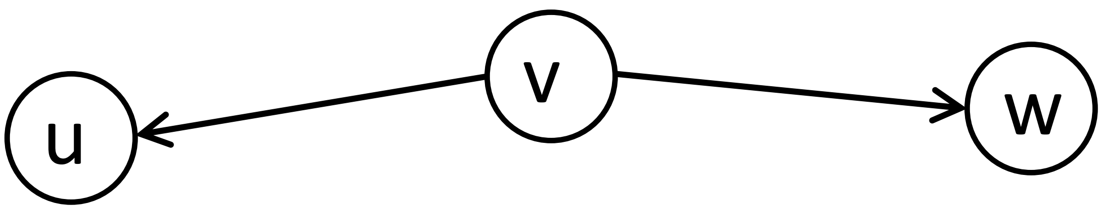
Problems

- Reachability no longer symmetric
  - can reach w from v but not v from w.
- Maybe allow reachability in either direction?



Problems

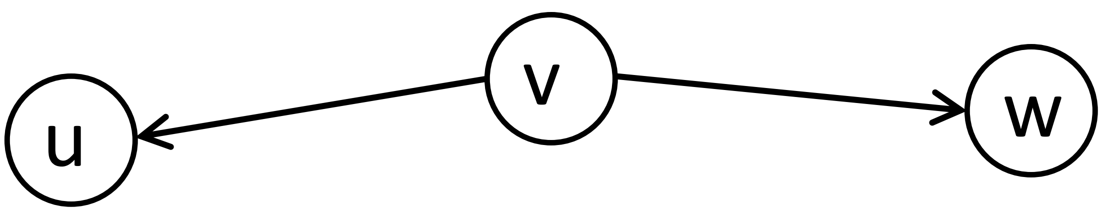
- Reachability no longer symmetric
  - can reach w from v but not v from w.
- Maybe allow reachability in either direction?



- Maybe allow you to follow edges in either direction?

Problems

- Reachability no longer symmetric
  - can reach w from v but not v from w.
- Maybe allow reachability in either direction?



- Maybe allow you to follow edges in either direction?
  - This basically treats digraph as undirected.

Strongly Connected Components

Definition: In a directed graph G, two vertices v and w are in the same Strongly Connected Component (SCC) if v is reachable from w *and* w is reachable from v.

Strongly Connected Components

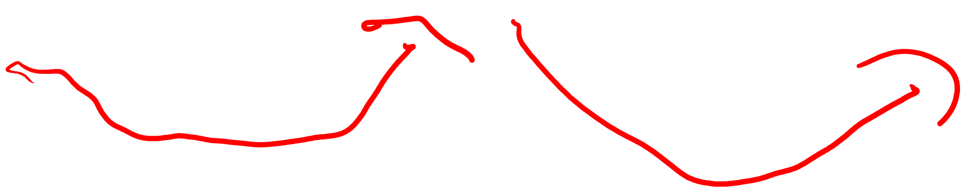
Definition: In a directed graph G, two vertices v and w are in the same Strongly Connected Component (SCC) if v is reachable from w *and* w is reachable from v.

Question: Can you actually partition the vertices into such components?

Equivalence Relation

Let v ~ w if v reachable from w and visa versa.

<u>Claim:</u> This is an <u>equivalence relation</u>. Namely:

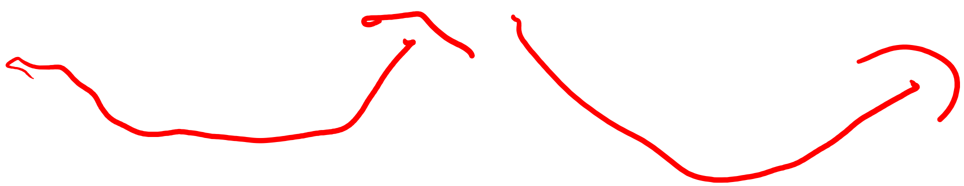


Equivalence Relation

Let v ~ w if v reachable from w and visa versa.

<u>Claim:</u> This is an <u>equivalence relation</u>. Namely:

- v ~ v. (v reachable from itself)

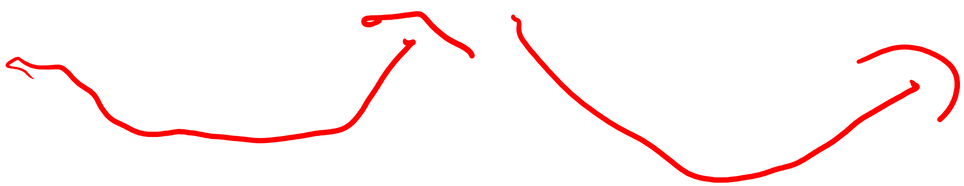


Equivalence Relation

Let v ~ w if v reachable from w and visa versa.

<u>Claim:</u> This is an <u>equivalence relation</u>. Namely:

- v ~ v. (v reachable from itself)
- If v ~ w then w ~ v. (relation is symmetric)

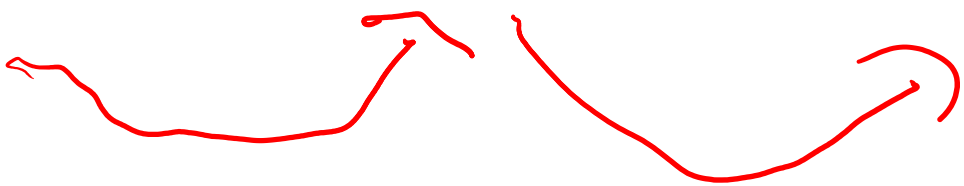


Equivalence Relation

Let v ~ w if v reachable from w and visa versa.

<u>Claim:</u> This is an <u>equivalence relation</u>. Namely:

- v ~ v. (v reachable from itself)
- If v ~ w then w ~ v. (relation is symmetric)
- If u ~ v and v ~ w then u ~ w.

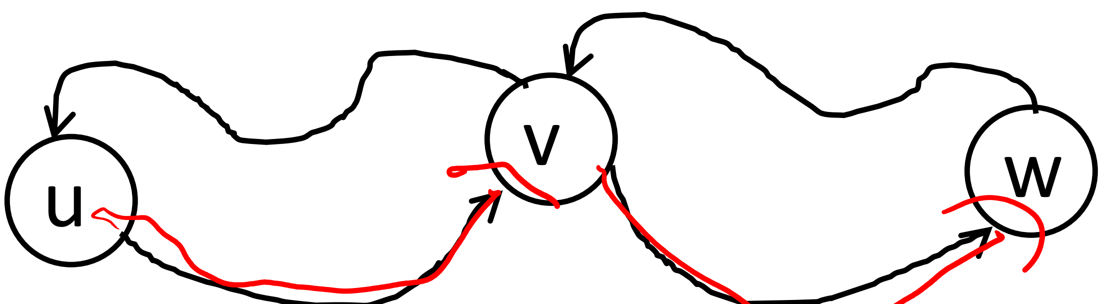


Equivalence Relation

Let v ~ w if v reachable from w and visa versa.

Claim: This is an equivalence relation. Namely:

- v ~ v. (v reachable from itself)
- If v ~ w then w ~ v. (relation is symmetric)
- If u ~ v and v ~ w then u ~ w.



Components

Whenever you have an equivalence relation you can split a set into components (equivalence classes) so that $v \sim w$ if and only if v and w are in the same component.

Components

Whenever you have an equivalence relation you can split a set into components (equivalence classes) so that $v \sim w$ if and only if v and w are in the same component.

Take any v, the set of all w so that $v \sim w$ is the component of v. Everything connects to everything else in this class and does not connect (both ways) to anything outside.

Question: SCCs

How many strongly connected components does the graph below have?

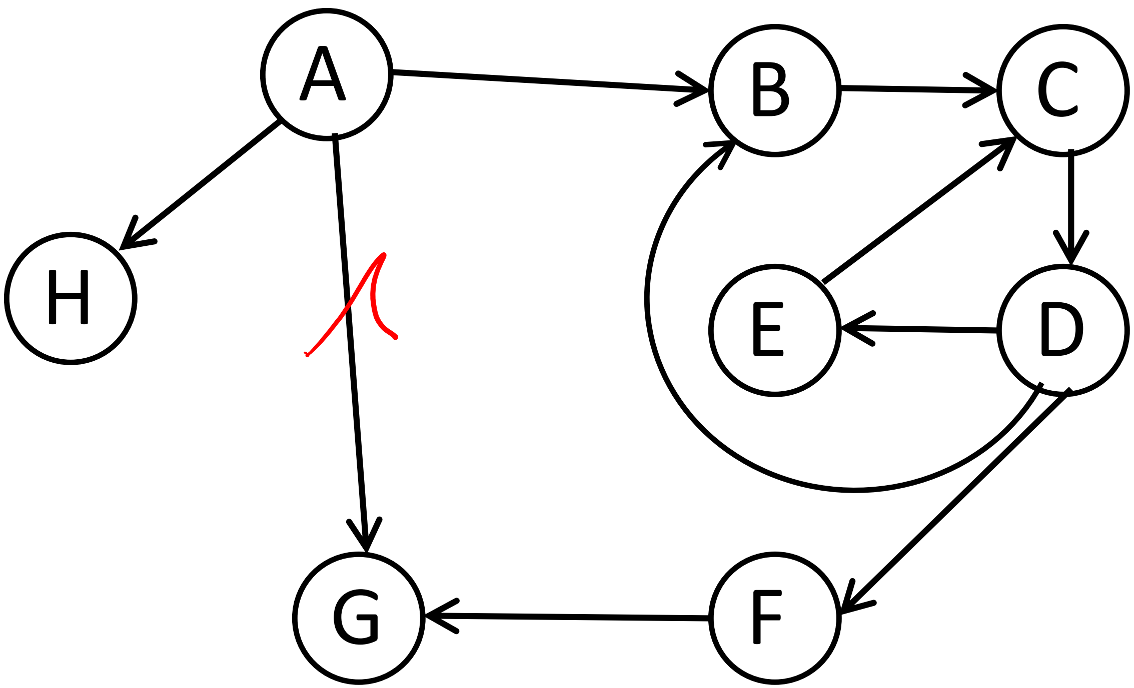
A) 1

B) 2

C) 3

D) 4

E) 5



Question: SCCs

How many strongly connected components does the graph below have?

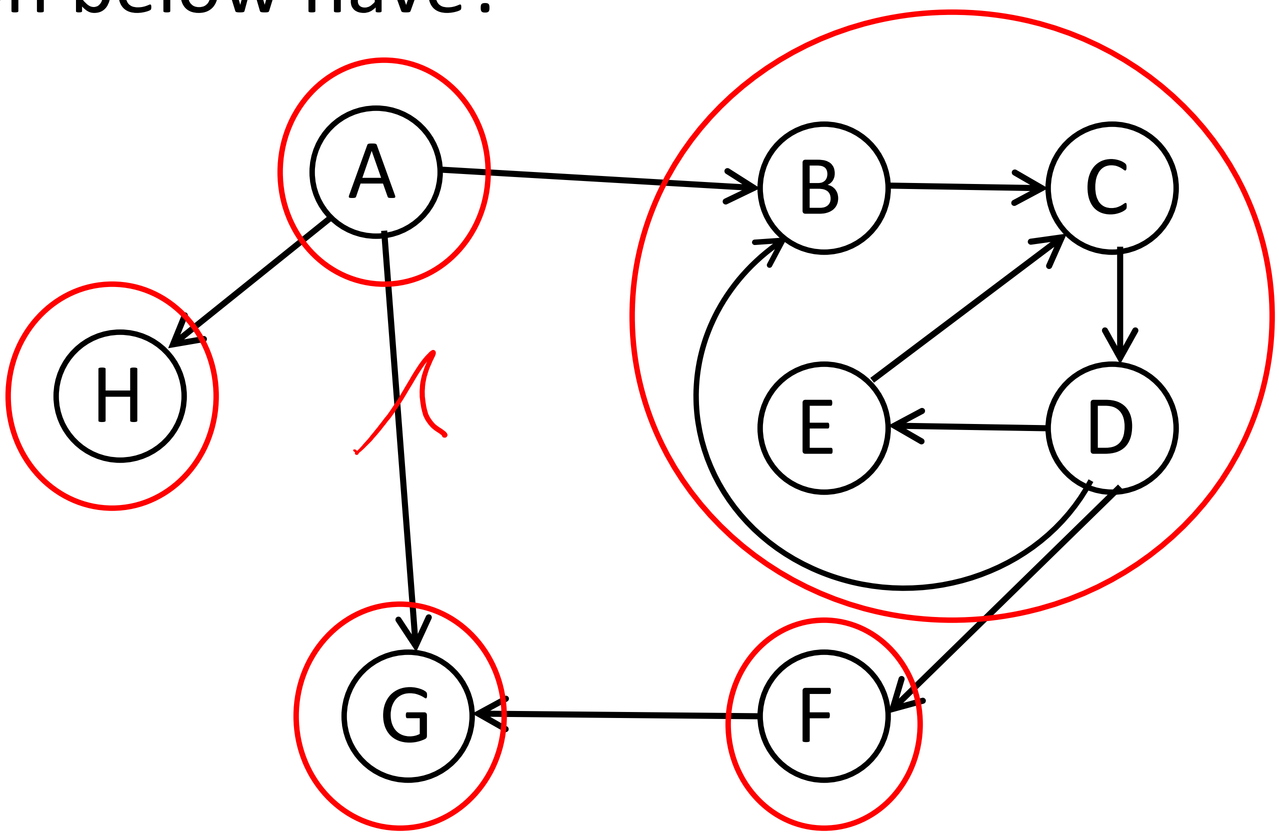
A) 1

B) 2

C) 3

D) 4

E) 5

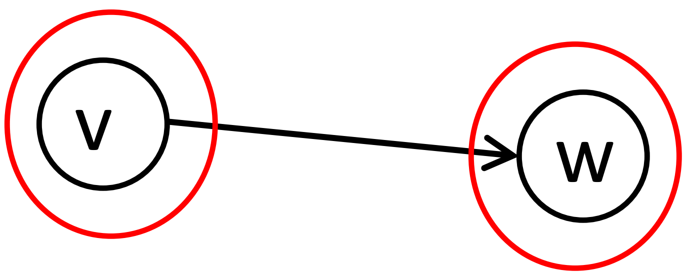


Connectivity

- Do strongly connected components completely describe connectivity in G?

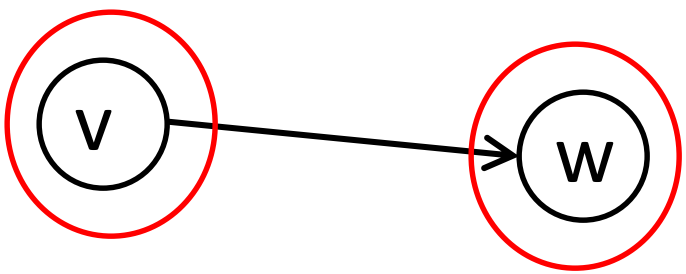
Connectivity

- Do strongly connected components completely describe connectivity in G?
  - No! In directed case, can have edges between SCCs.



Connectivity

- Do strongly connected components completely describe connectivity in G?
  - No! In directed case, can have edges between SCCs.

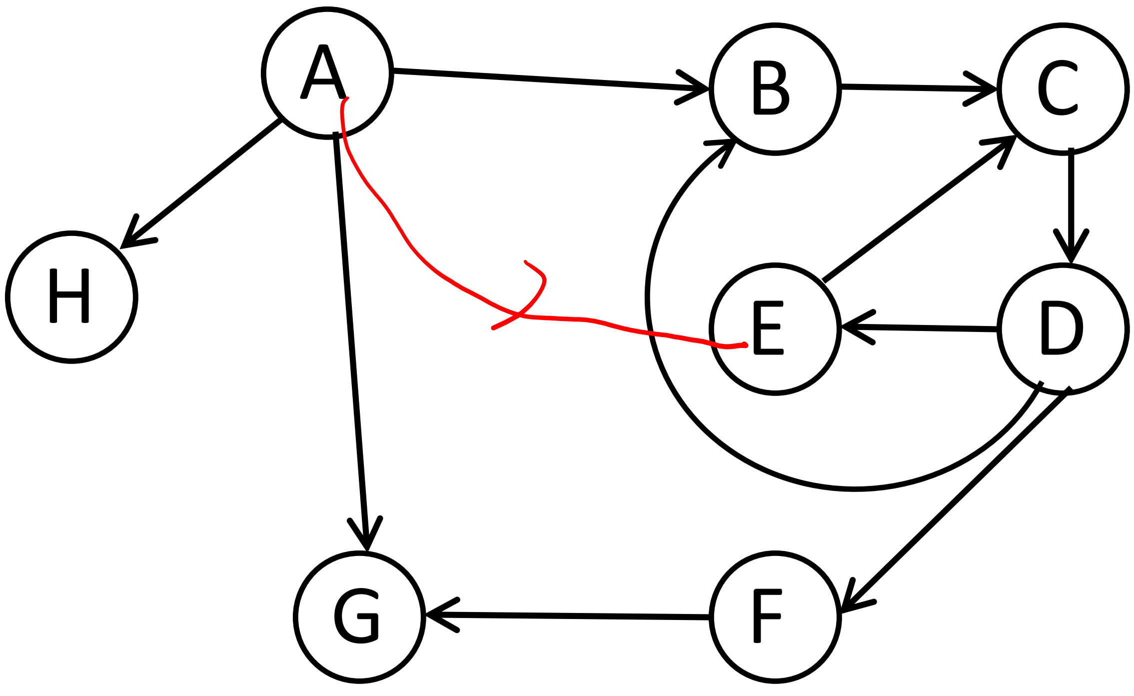


- Need extra information to describe how SCCs connect.

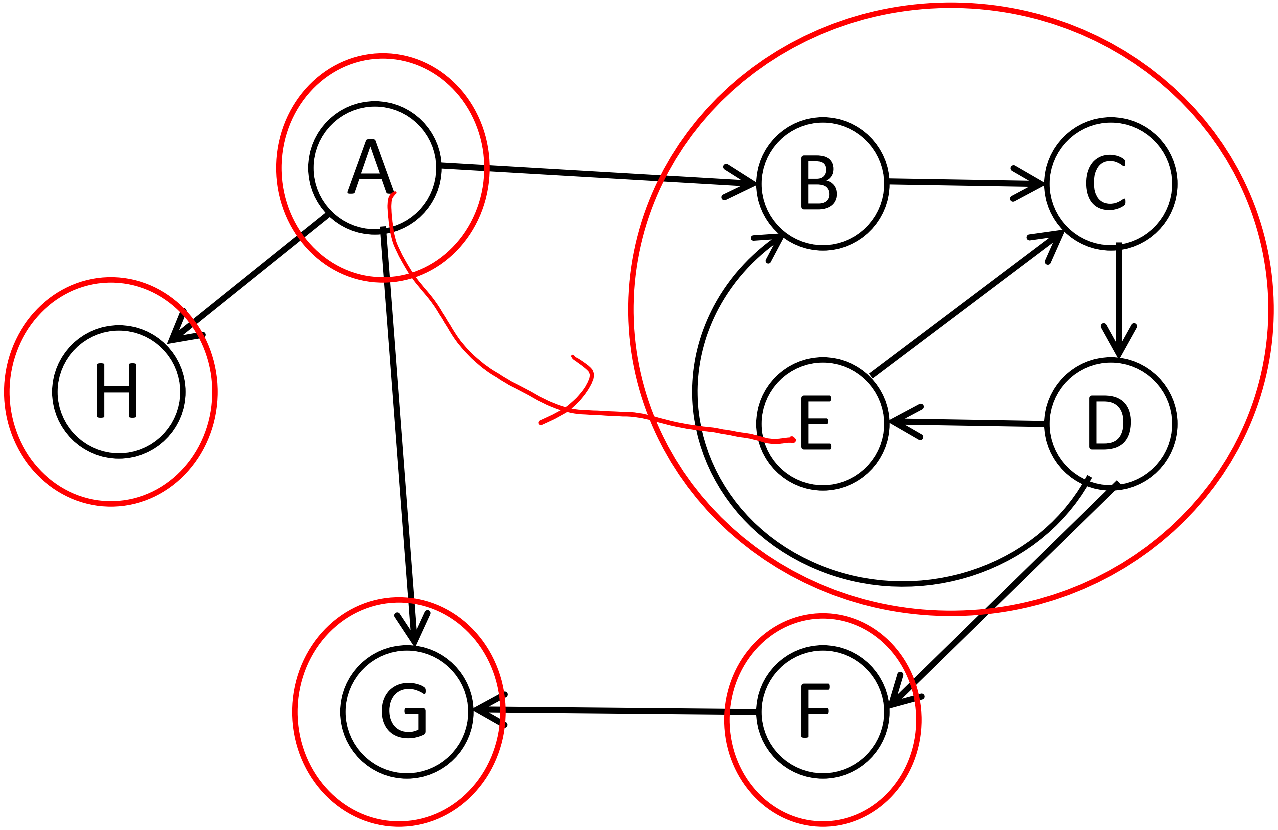
Metagraph

Definition: The metagraph of a directed graph G is a graph whose vertices are the SCCs of G, where there is an edge between C_1 and C_2 if and only if G has an edge between some vertex of C_1 and some vertex of C_2.

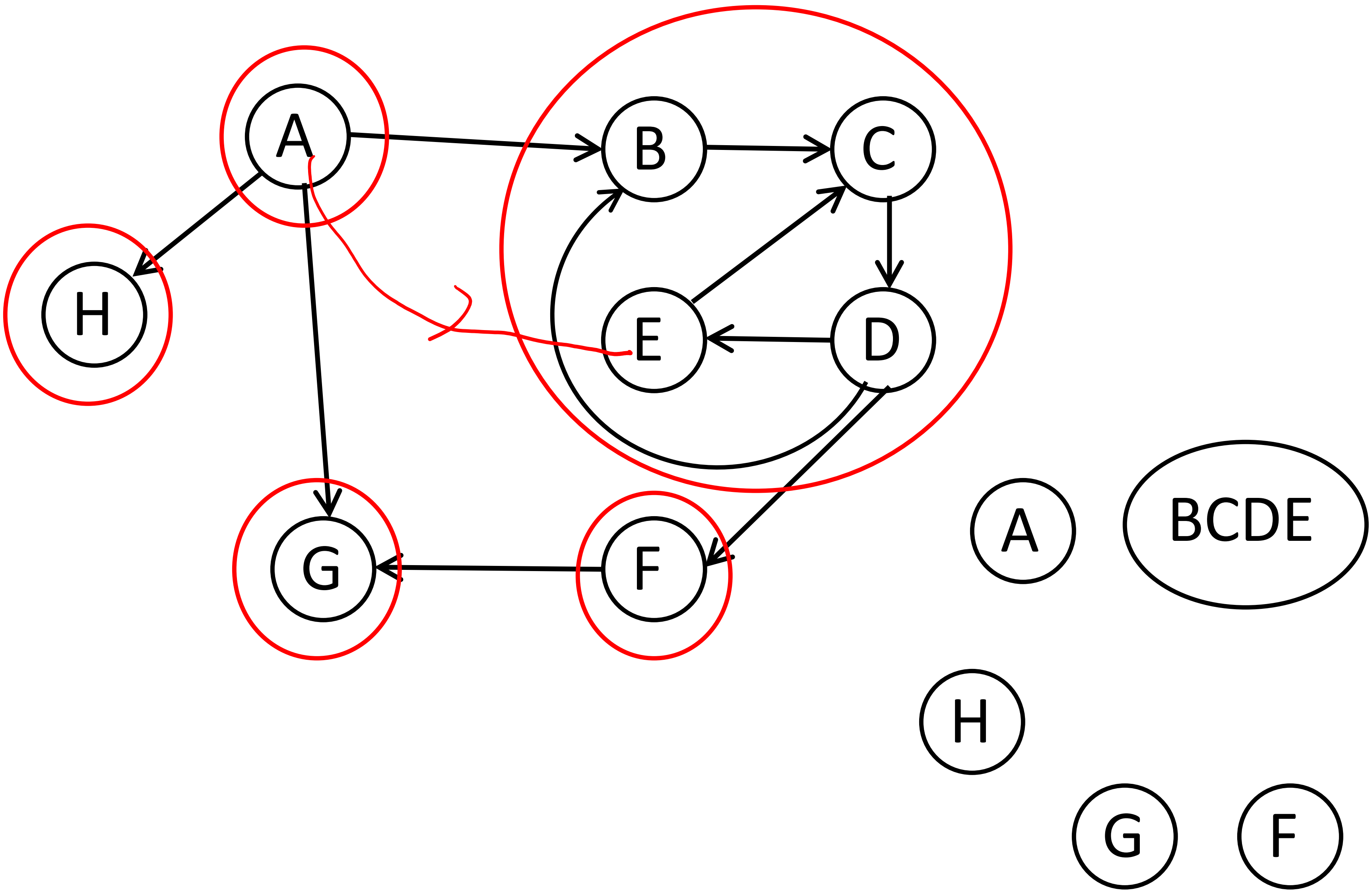
Example



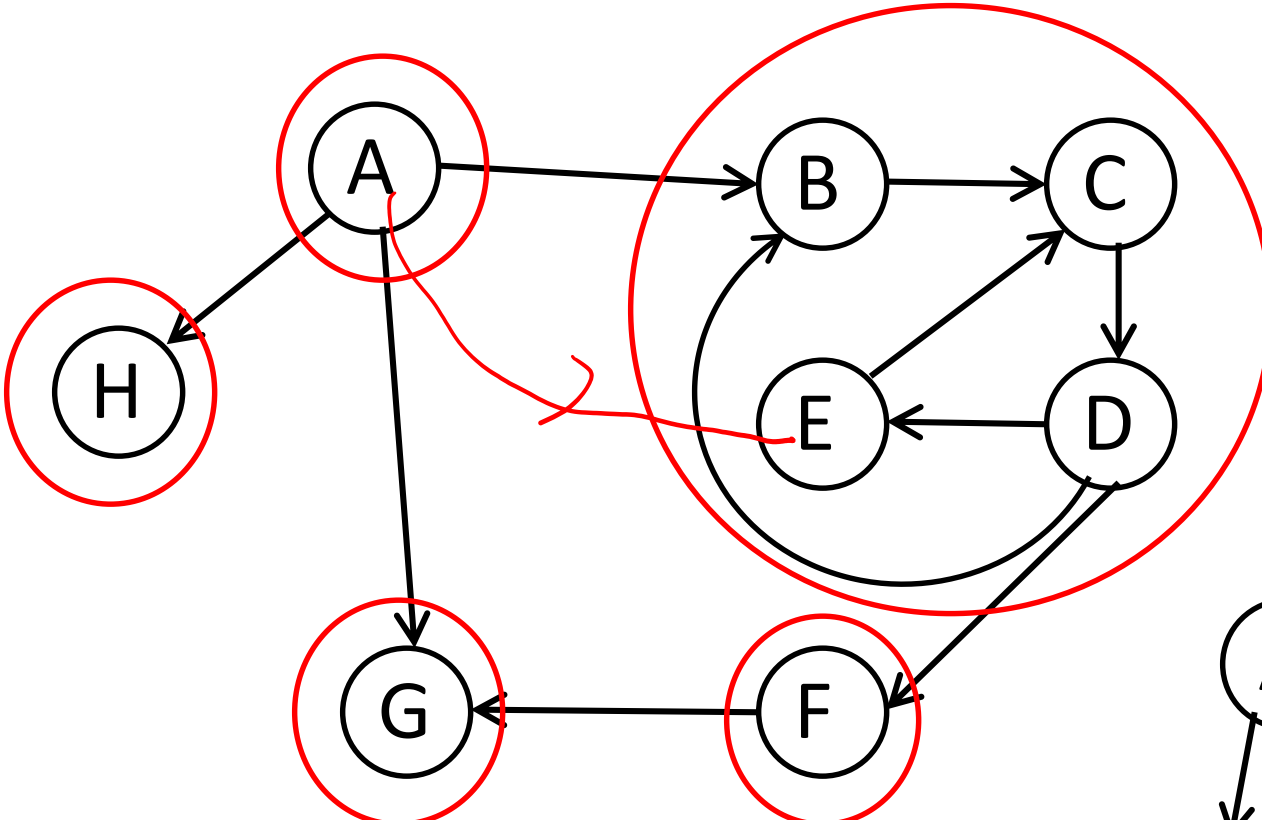
Example

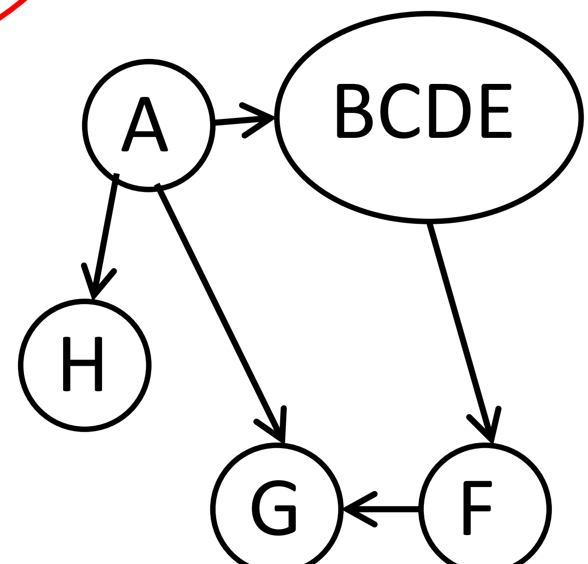


Example

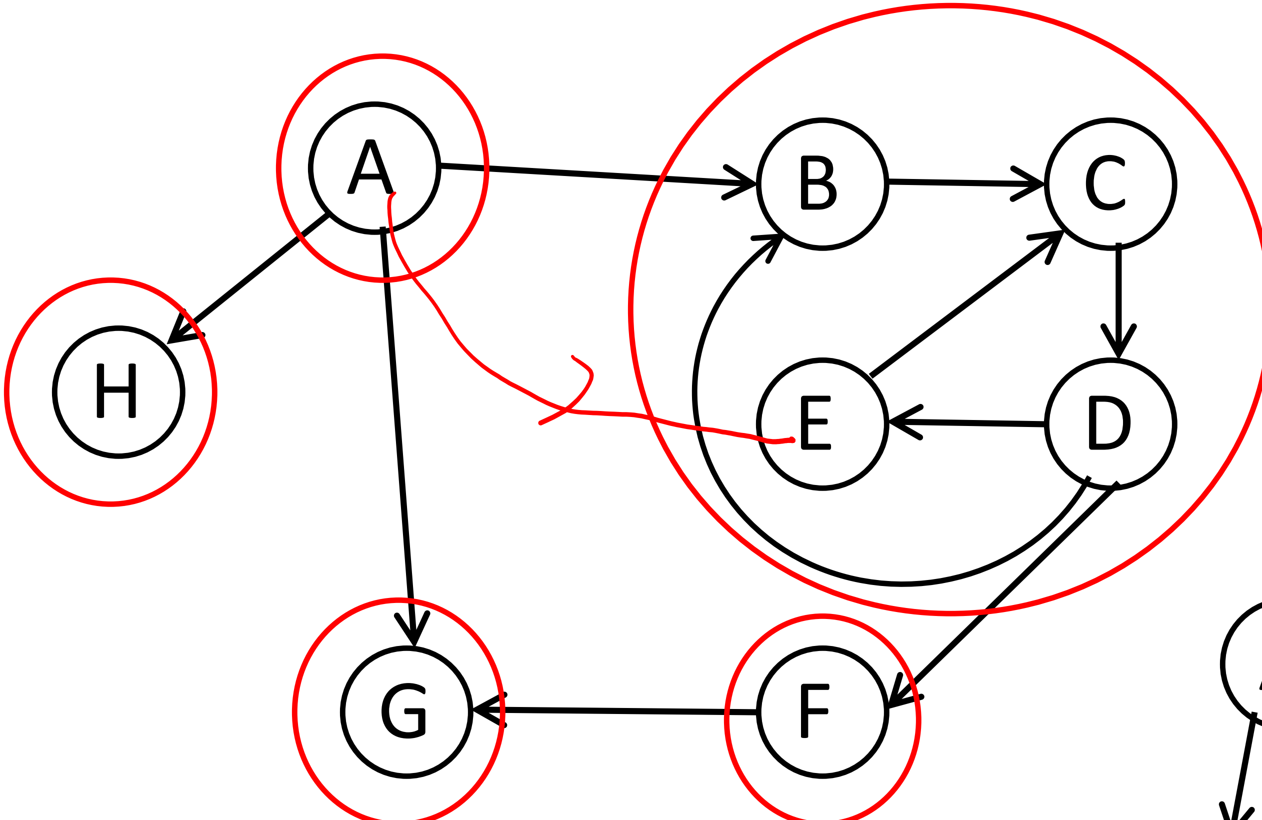


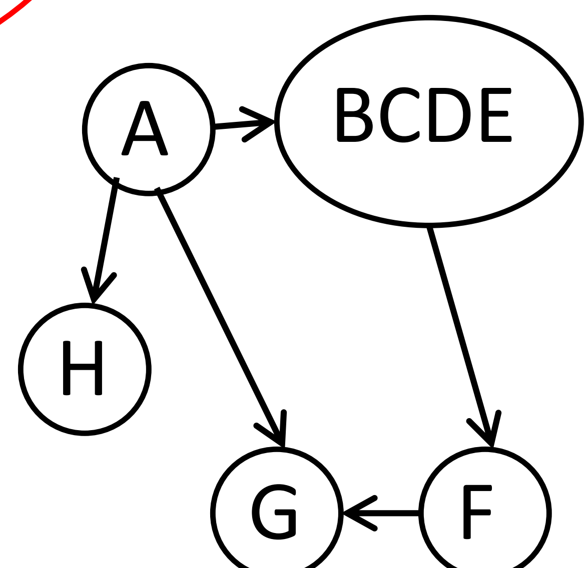
Example





Example





Note: Metagraph is a DAG!

Result

Theorem: The metagraph is any directed graph is a DAG.

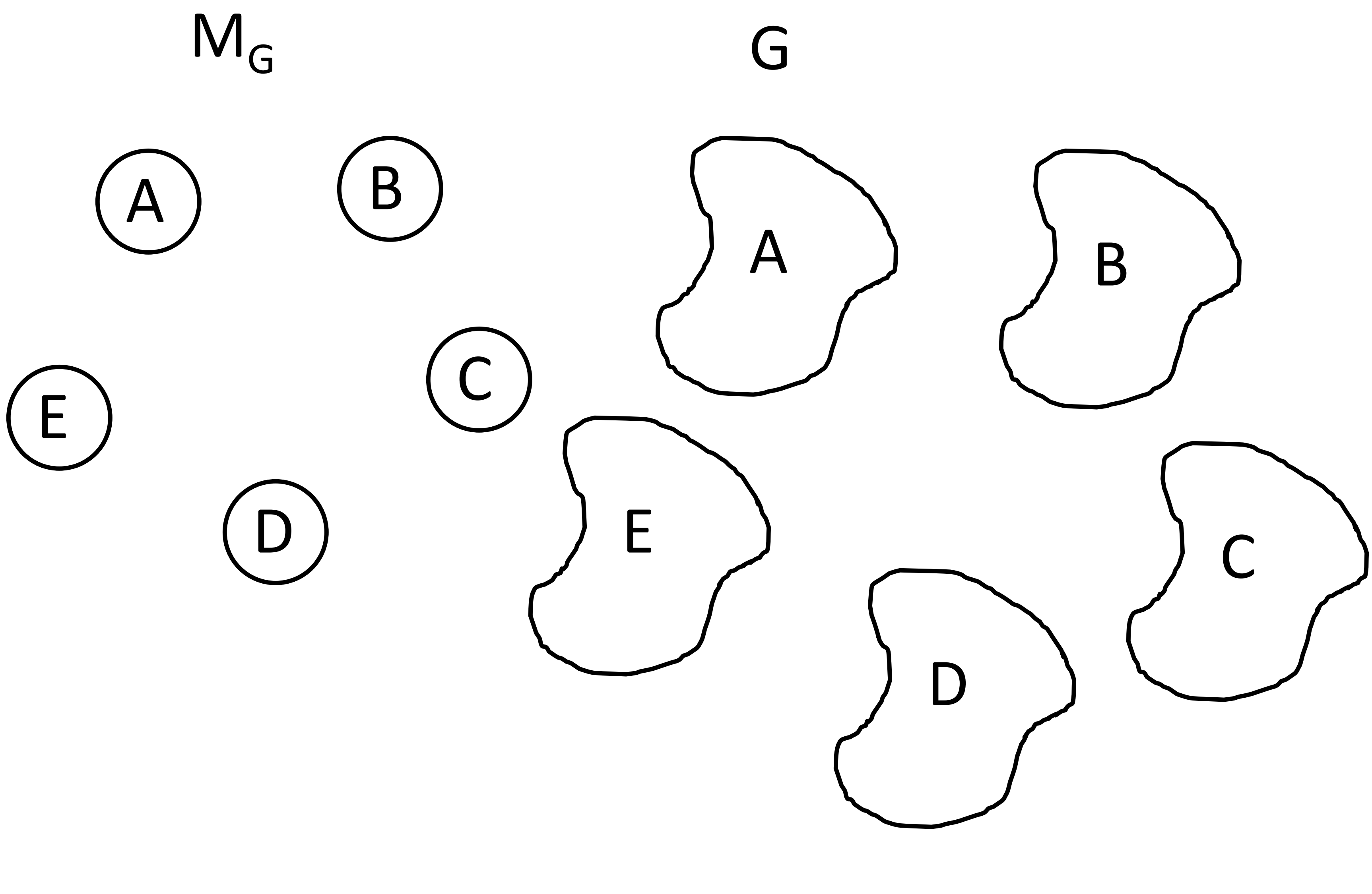
Result

Theorem: The metagraph is any directed graph is a DAG.

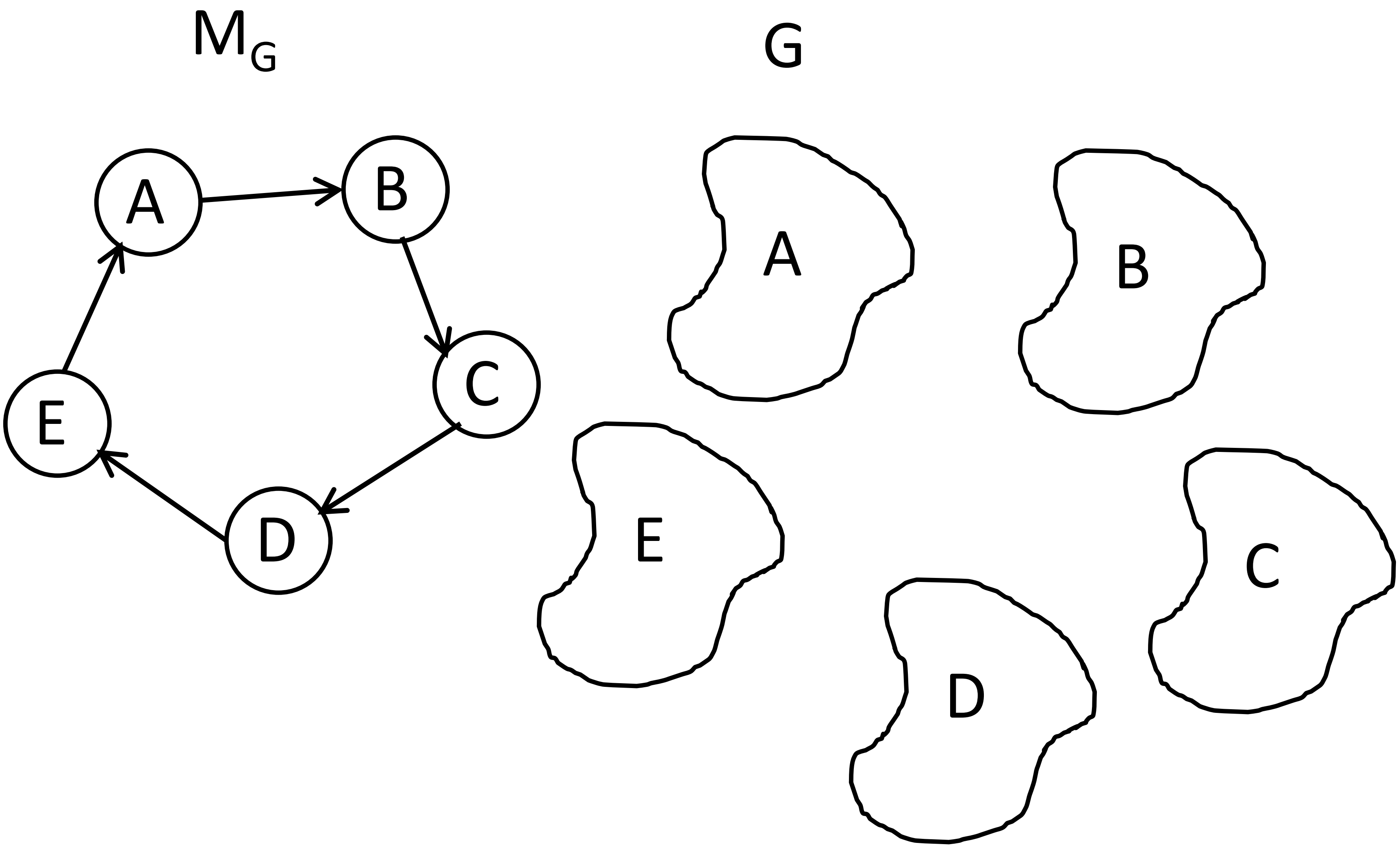
Proof (sketch):

- Assume for sake of contradiction it is not.
- Then metagraph has a cycle.
- Use this to show that separated components should be connected.

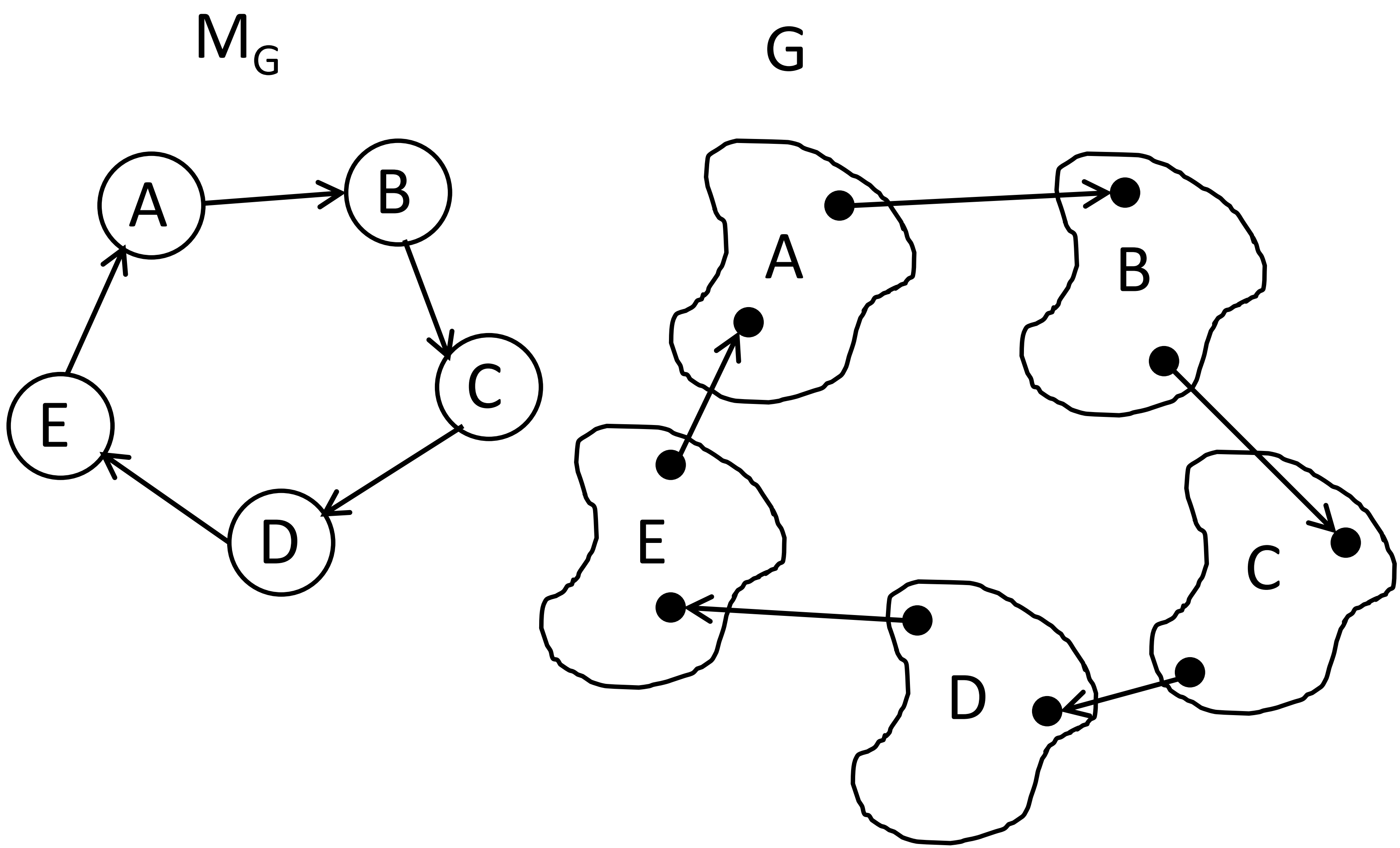
Proof



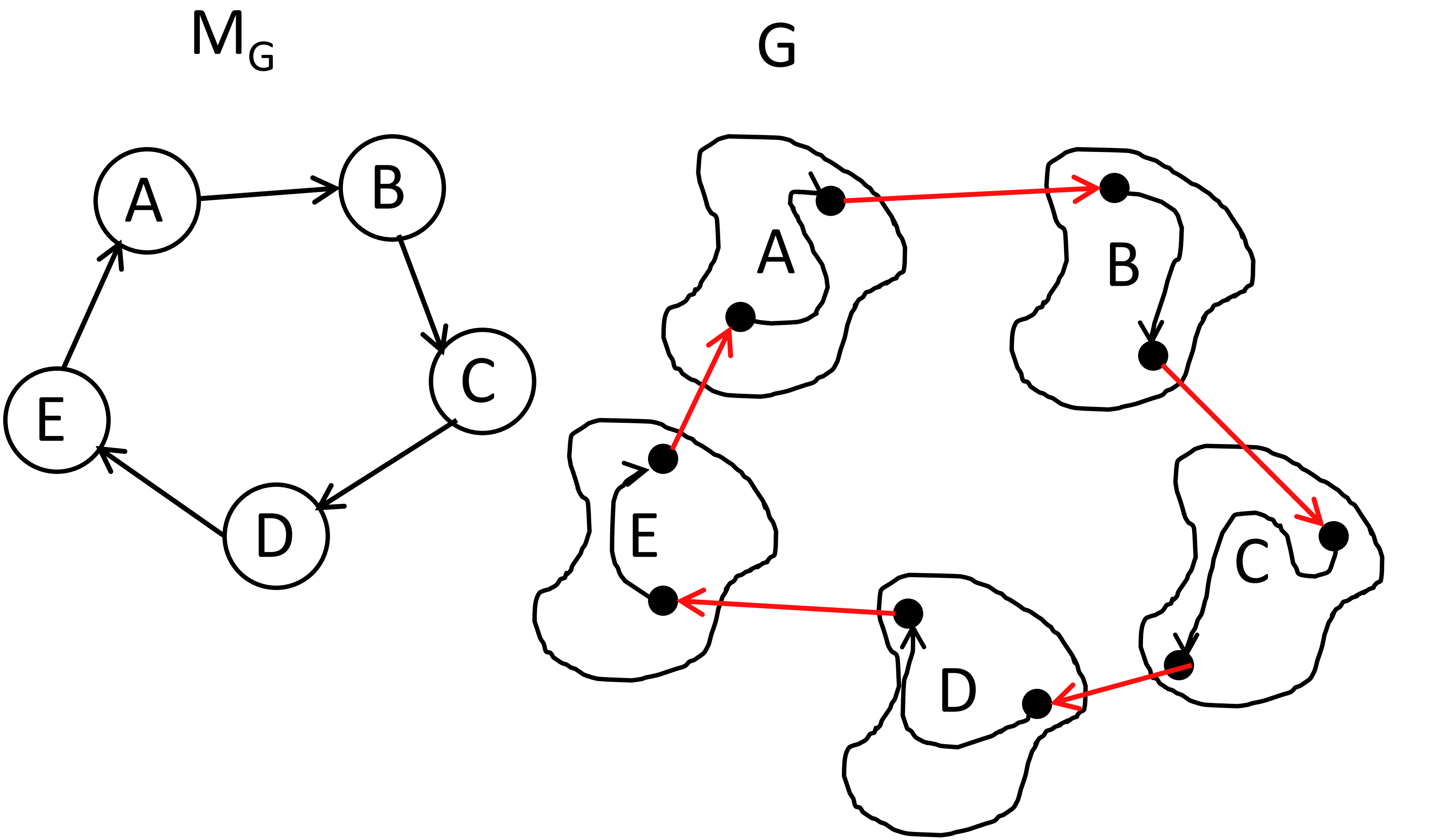
Proof



Proof



Proof



Computing SCCs

Problem: Given a directed graph G compute the SCCs of G and its metagraph.

Computing SCCs

Problem: Given a directed graph G compute the SCCs of G and its metagraph.

Easy Algorithm:

- For each v compute vertices reachable from v.
- Find pairs v, w so that v reachable from w and visa versa.
- For each v the corresponding w's are the SCC of v.

Computing SCCs

Problem: Given a directed graph G compute the SCCs of G and its metagraph.

Easy Algorithm:

- For each v compute vertices reachable from v.
- Find pairs v, w so that v reachable from w and visa versa.
- For each v the corresponding w's are the SCC of v.

Runtime: $O(|V|(|V|+|E|))$. We can do better.

Observation

Suppose that SCC(v) is a sink in the metagraph.

Observation

Suppose that SCC(v) is a sink in the metagraph.

- G has no edges from SCC(v) to another SCC.

Observation

Suppose that SCC(v) is a sink in the metagraph.

- G has no edges from SCC(v) to another SCC.
- Run `explore(v)` to find all vertices reachable from v.

Observation

Suppose that SCC(v) is a sink in the metagraph.

- G has no edges from SCC(v) to another SCC.
- Run `explore(v)` to find all vertices reachable from v.
  - Contains all vertices in SCC(v).

Observation

Suppose that SCC(v) is a sink in the metagraph.

- G has no edges from SCC(v) to another SCC.
- Run `explore(v)` to find all vertices reachable from v.
  - Contains all vertices in SCC(v).
  - Contains no other vertices.

Observation

Suppose that SCC(v) is a sink in the metagraph.

- G has no edges from SCC(v) to another SCC.
- Run `explore(v)` to find all vertices reachable from v.
  - Contains all vertices in SCC(v).
  - Contains no other vertices.
- If v in sink SCC, `explore(v)` finds *exactly* v's component.