

Searching & Sorting Graphs

Today: directed graphs

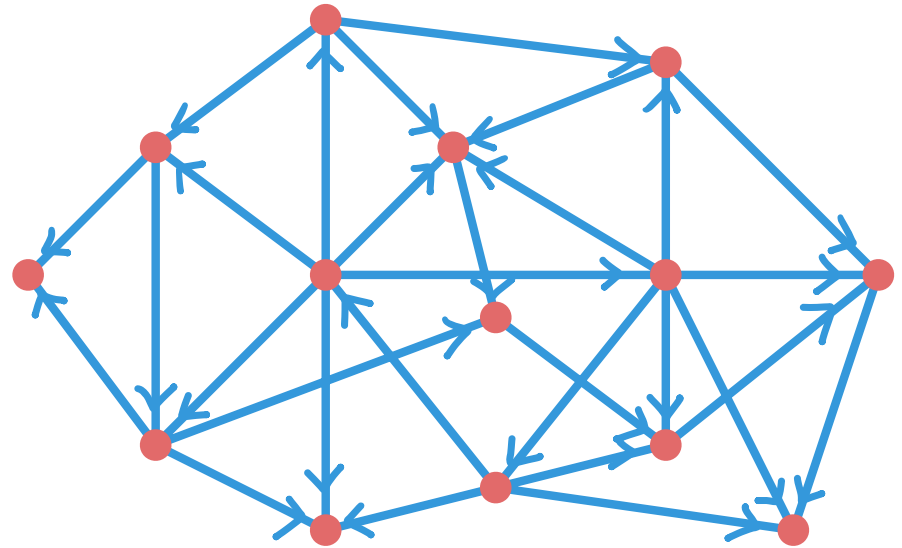
acyclic graphs,

topological sort,

cycles, reachability,

depth-first search,

strongly connected components



(which give a lot of structure to a very simple and broadly applicable model)

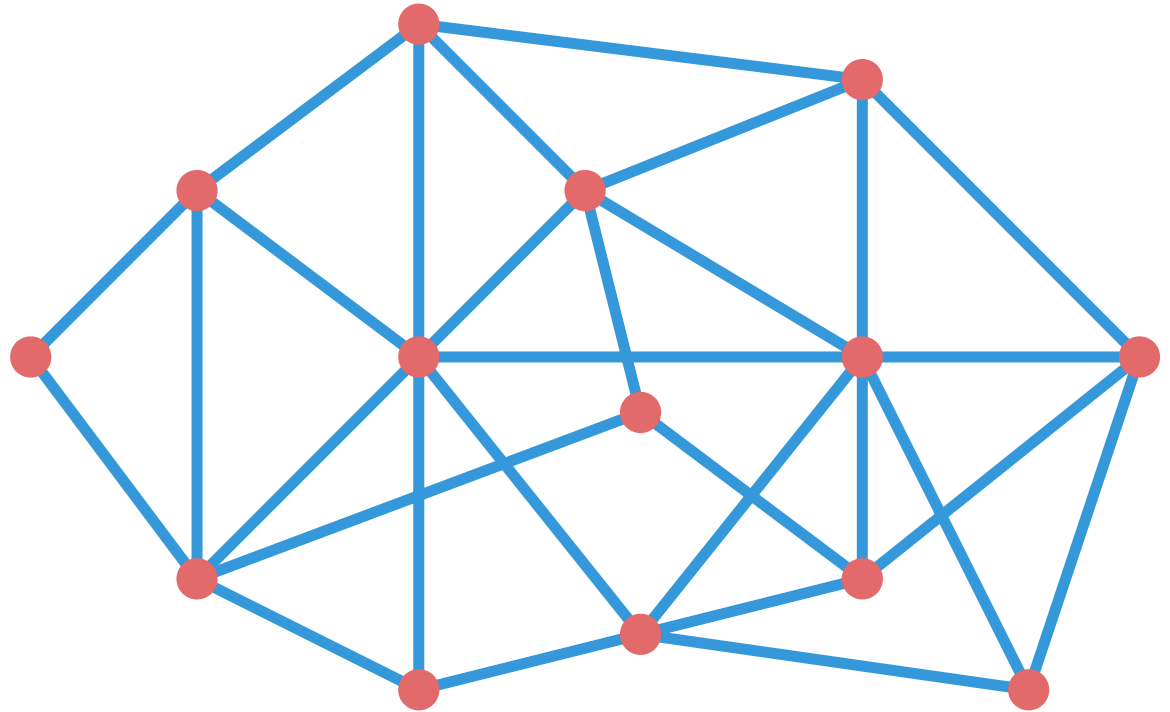
Graphs

$$G=(V, E)$$

$$n=|V|, m=|E|$$

vertices •

edges —————

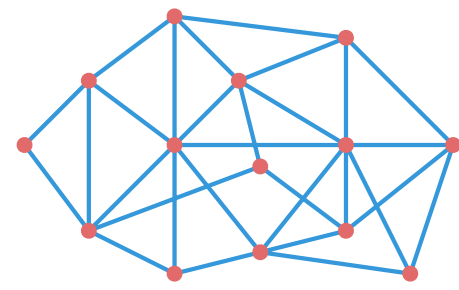


undirected
edge
between u and v



directed
edge
from u to v

Representing graphs



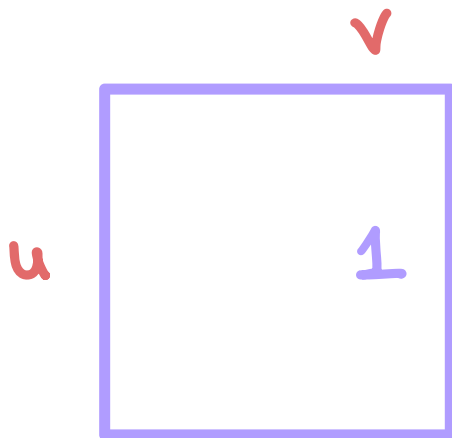
① "Adjacency list"

for $v \in V$, list all (incoming, outgoing) edges of v

e.g. $v.out = [a, b, c, \dots]$, $v.in = [d, e, f, \dots]$

② "Adjacency matrix"

$A \in \mathbb{R}^{V \times V}$ w/ $A(u, v) = \text{weight of edge from } u \text{ to } v$
(=0 if no edge)



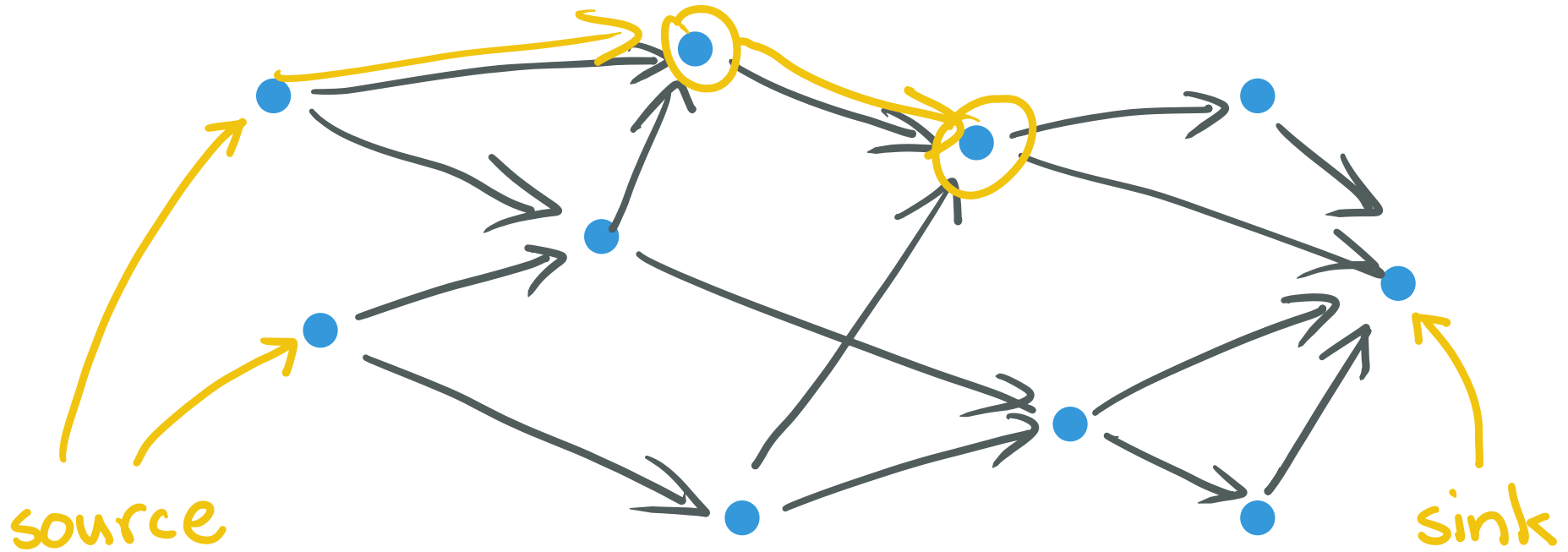
I. DAG's and topological orderings

II. Searching graphs

III. Sorting graphs

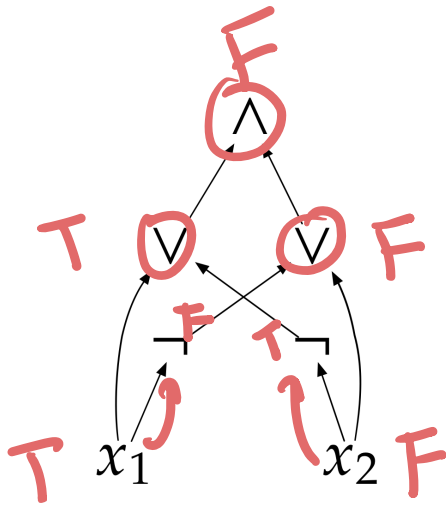
Directed Acyclic Graphs (DAG's)

(no cycles)

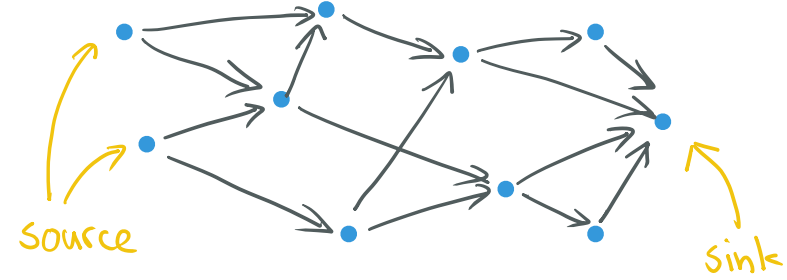


e.g. circuits

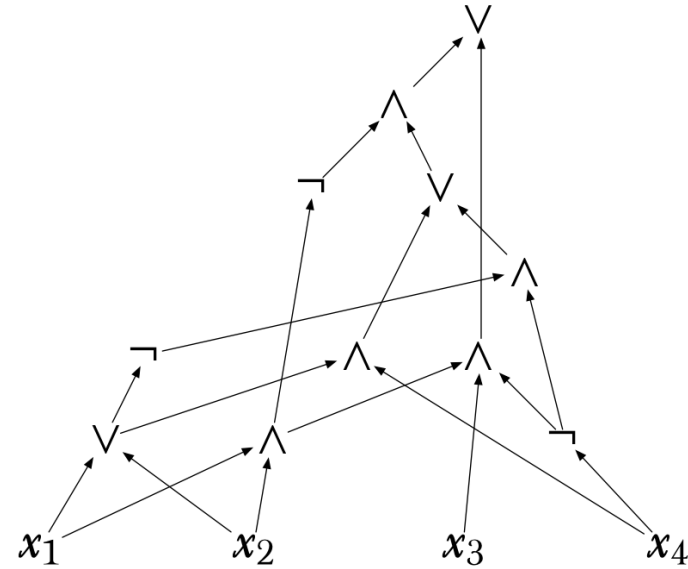
XOR ("exclusive-or")
(TRUE iff $x_1 \neq x_2$)



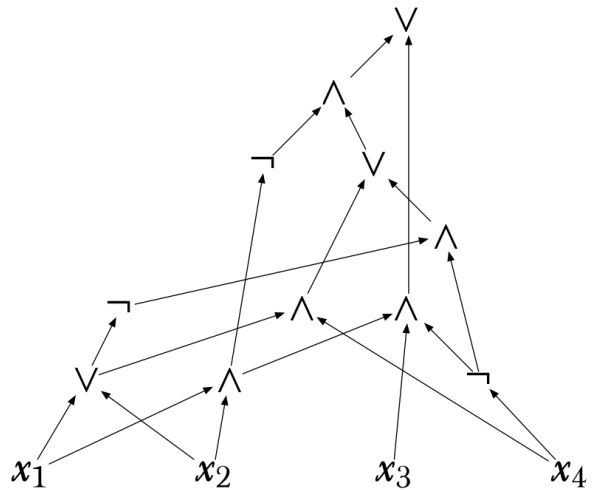
$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$



Addition



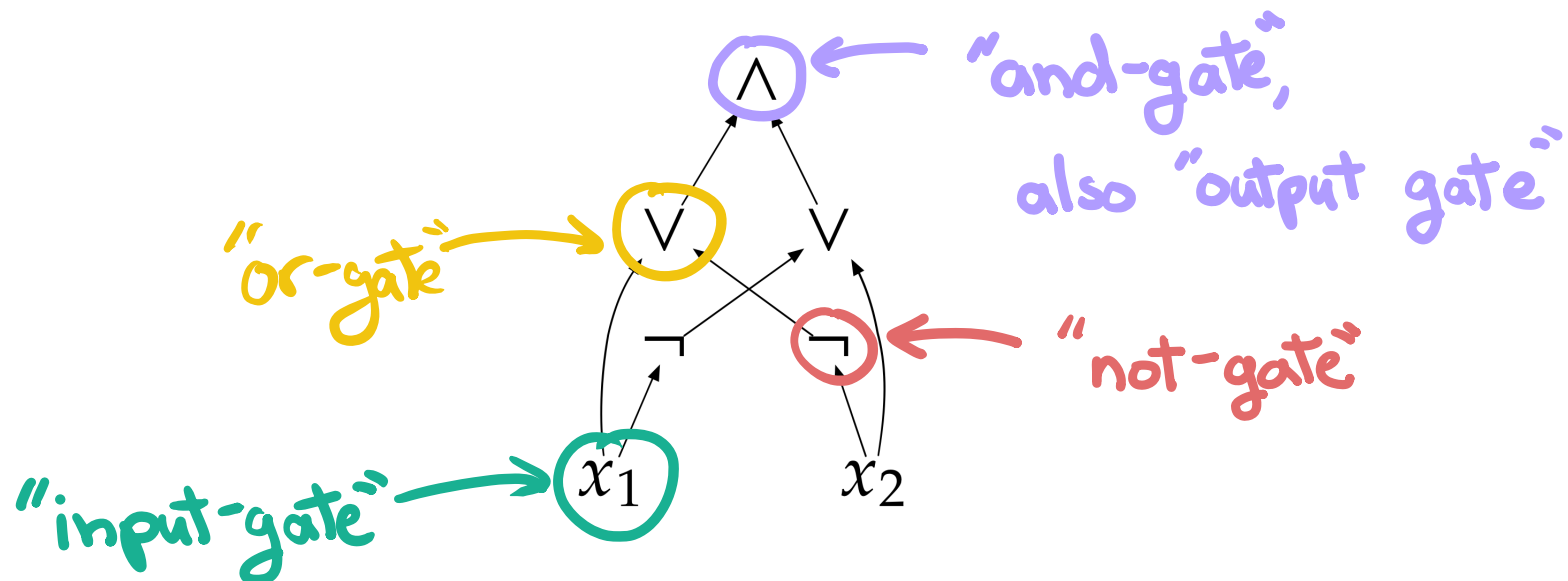
$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{if } x_1 + x_2 = x_3 x_4 \\ 0 & \text{otherwise} \end{cases}$$



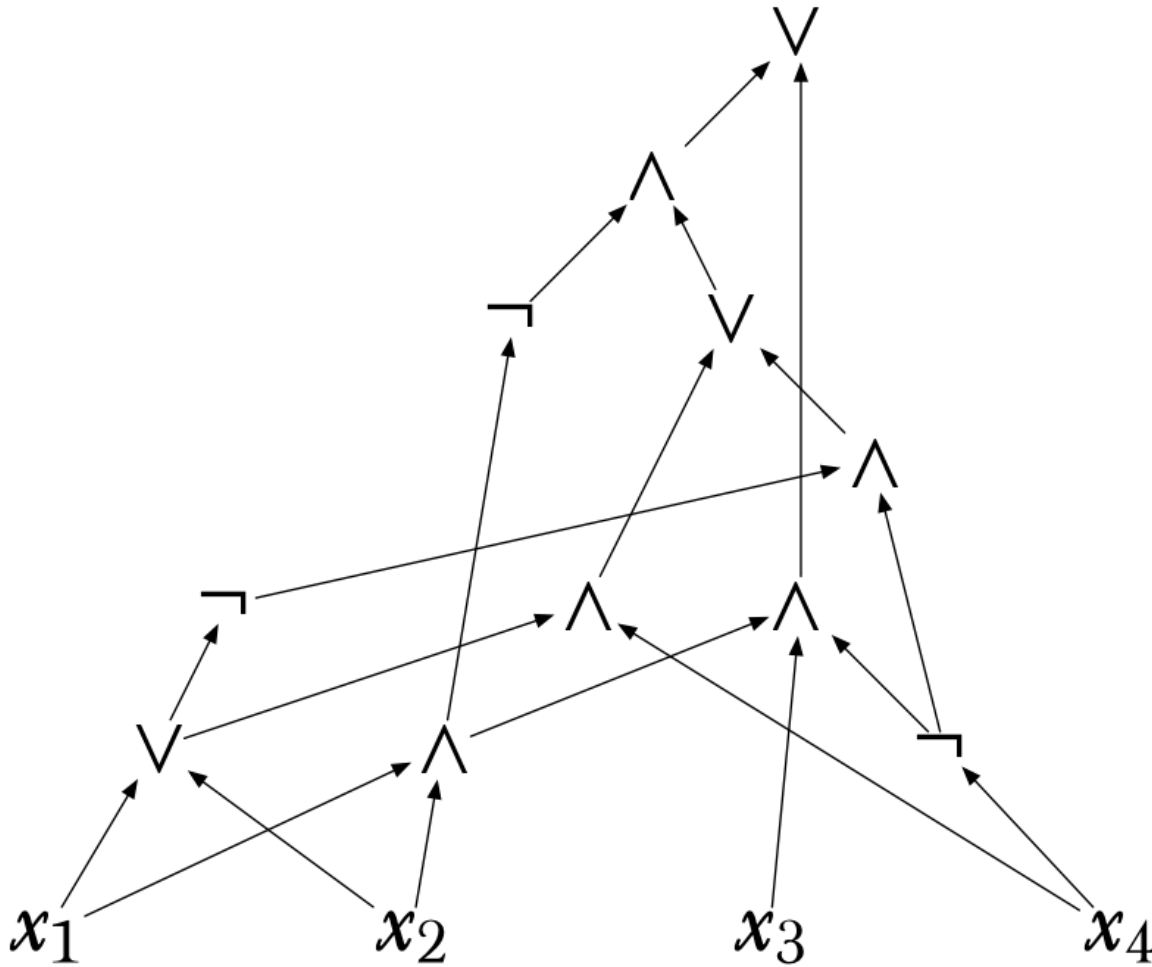
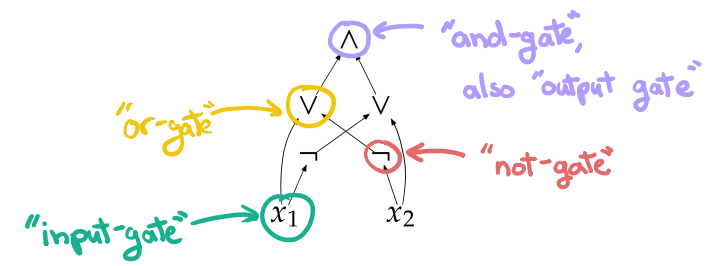
(definition)

logical gates arranged as vertices
in a directed acyclic graph

"gates": take some inputs bits, output 1 bit



How to evaluate a circuit?



(conceptually)

"bottom up,
one at a time"

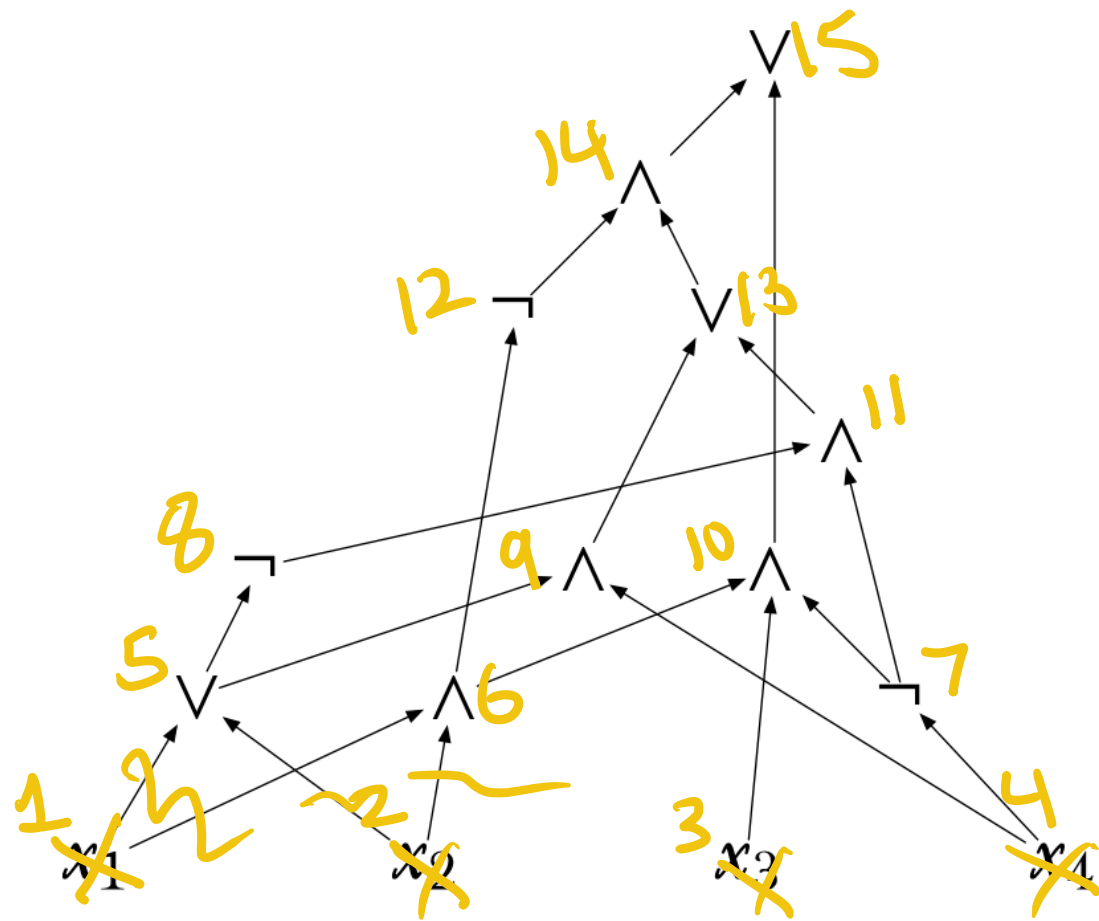
Topological order

think of arcs as " $\leq$ "

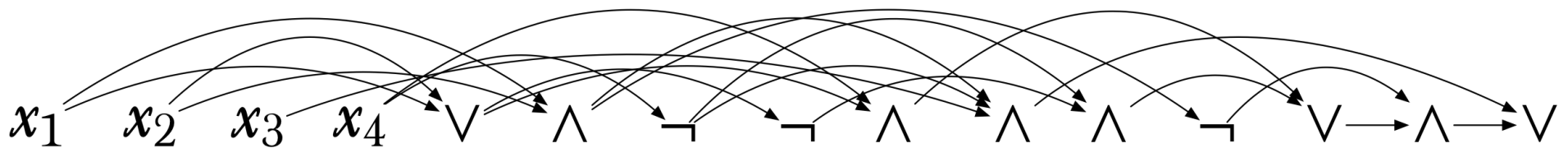
(directed edge)

$u \leq v$

this defines a partial order on V



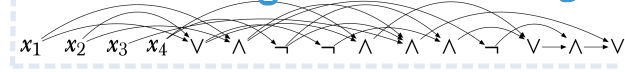
Topological ordering (of V):
any ordering consistent w/ edges



to evaluate, process gates from left to right!

How to topologically sort:
repeatedly remove sources

Topological ordering (of V):
any ordering consistent w/ edges

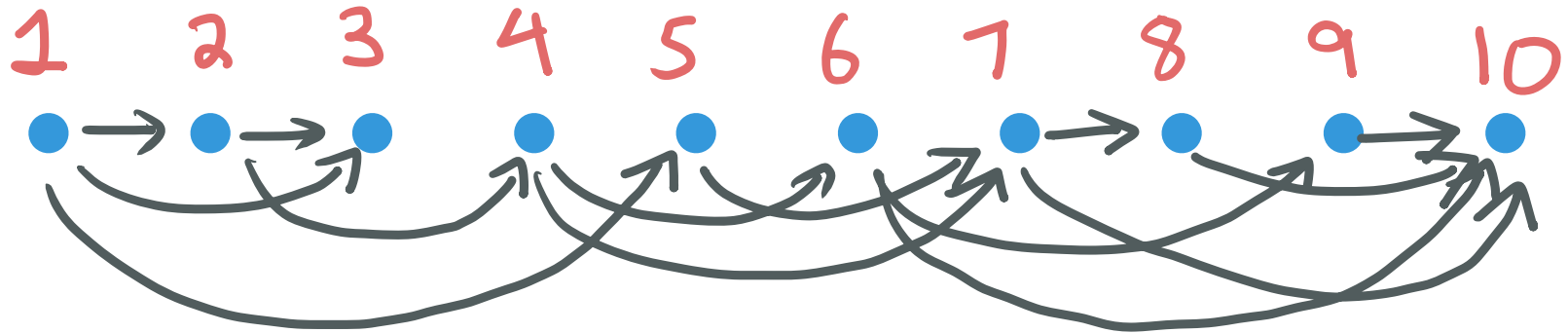
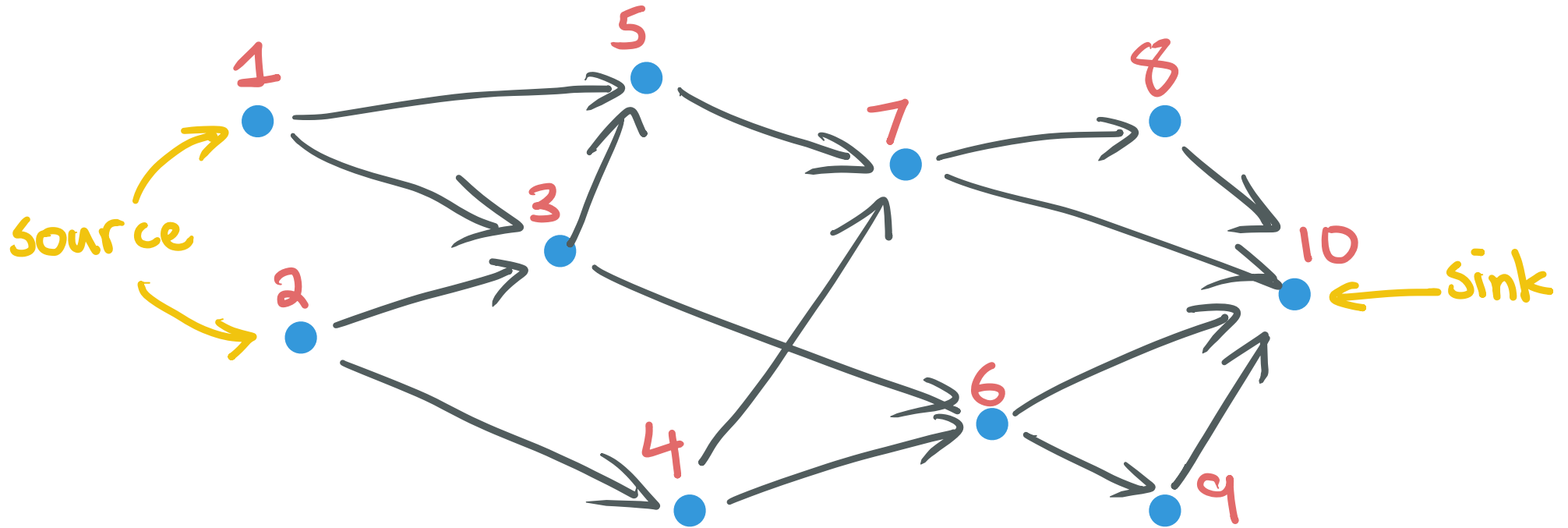
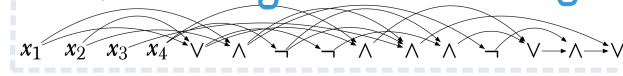


source

← sink

How to topologically sort:
repeatedly remove sources

Topological ordering (of V):
any ordering consistent w/ edges



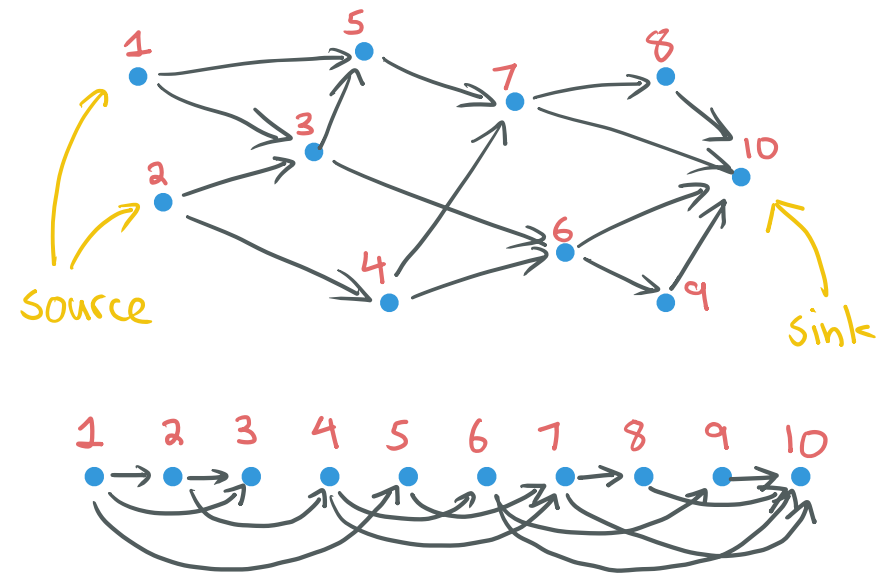
$O(m + n \log n)$

source-peeling($G = (V, E)$)

/ We assume G is a DAG. ↙ min-deg rel*

1. While there is a source $v \in V$
 - A. Append v to the output list.
 - B. Remove v and all incident edges from G .

↑ easy to make linear time by keeping track of in-degrees

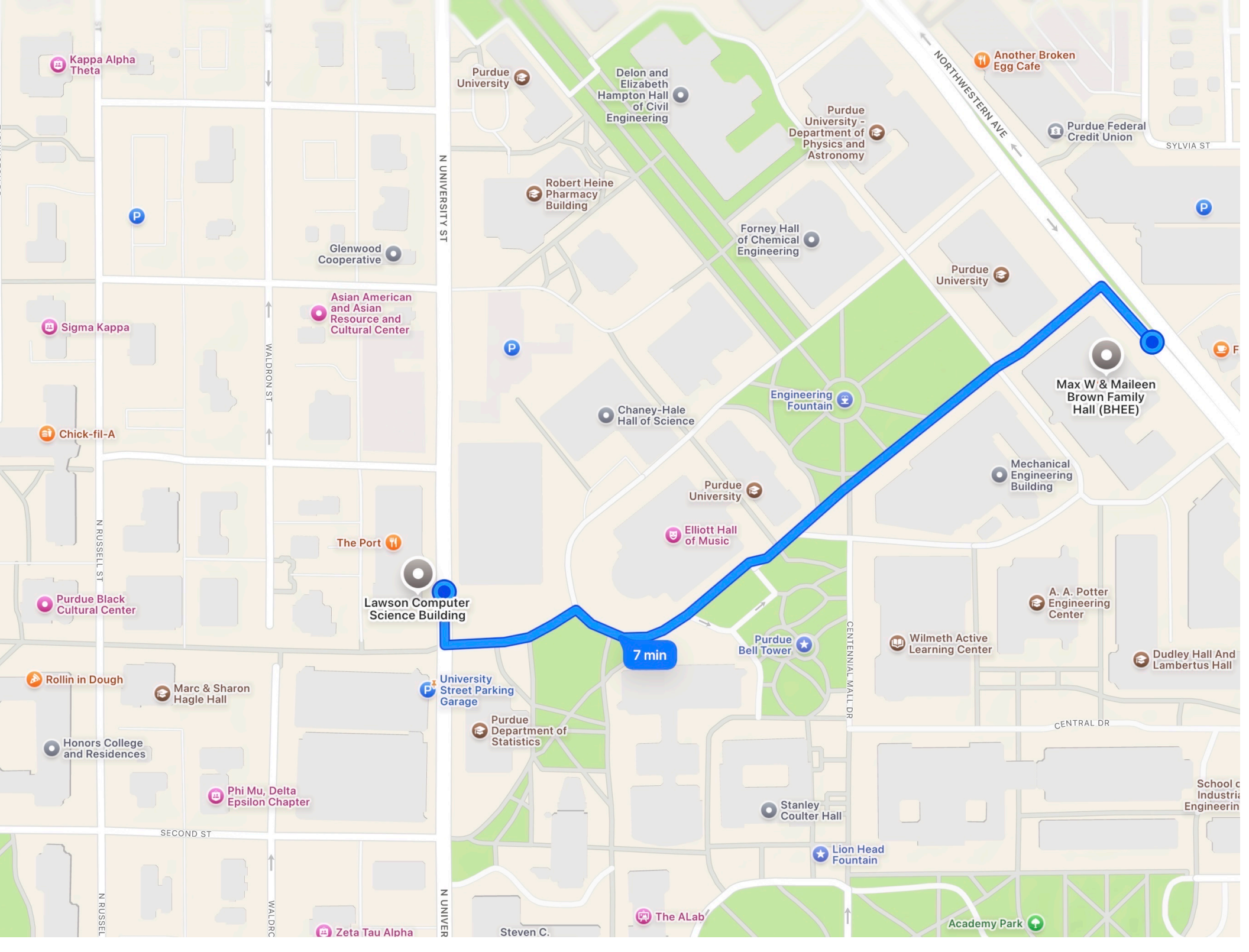


Theorem 6.1. Given a DAG G with m edges and n vertices, one can compute a topological ordering of V in $O(m + n)$ time.

I. DAG's and topological orderings

II. Searching graphs

III. Sorting graphs



Kappa Alpha Theta

Sigma Kappa

Chick-fil-A

Purdue Black Cultural Center

Rollin in Dough

Honors College and Residences

Phi Mu, Delta Epsilon Chapter

Zeta Tau Alpha

Asian American and Asian Resource and Cultural Center

Glenwood Cooperative

The Port

Lawson Computer Science Building

University Street Parking Garage

Purdue Department of Statistics

Purdue University

Robert Heine Pharmacy Building

Delon and Elizabeth Hampton Hall of Civil Engineering

Chaney-Hale Hall of Science

Elliott Hall of Music

Purdue University

Purdue Bell Tower

Forney Hall of Chemical Engineering

Purdue University - Department of Physics and Astronomy

The ALab

Stanley Coulter Hall

Lion Head Fountain

Wilmeth Active Learning Center

A. A. Potter Engineering Center

Mechanical Engineering Building

Max W & Maileen Brown Family Hall (BHEE)

Purdue Federal Credit Union

Another Broken Egg Cafe

Academy Park

Dudley Hall And Lambertus Hall

School of Industrial Engineering

CAT MAZE

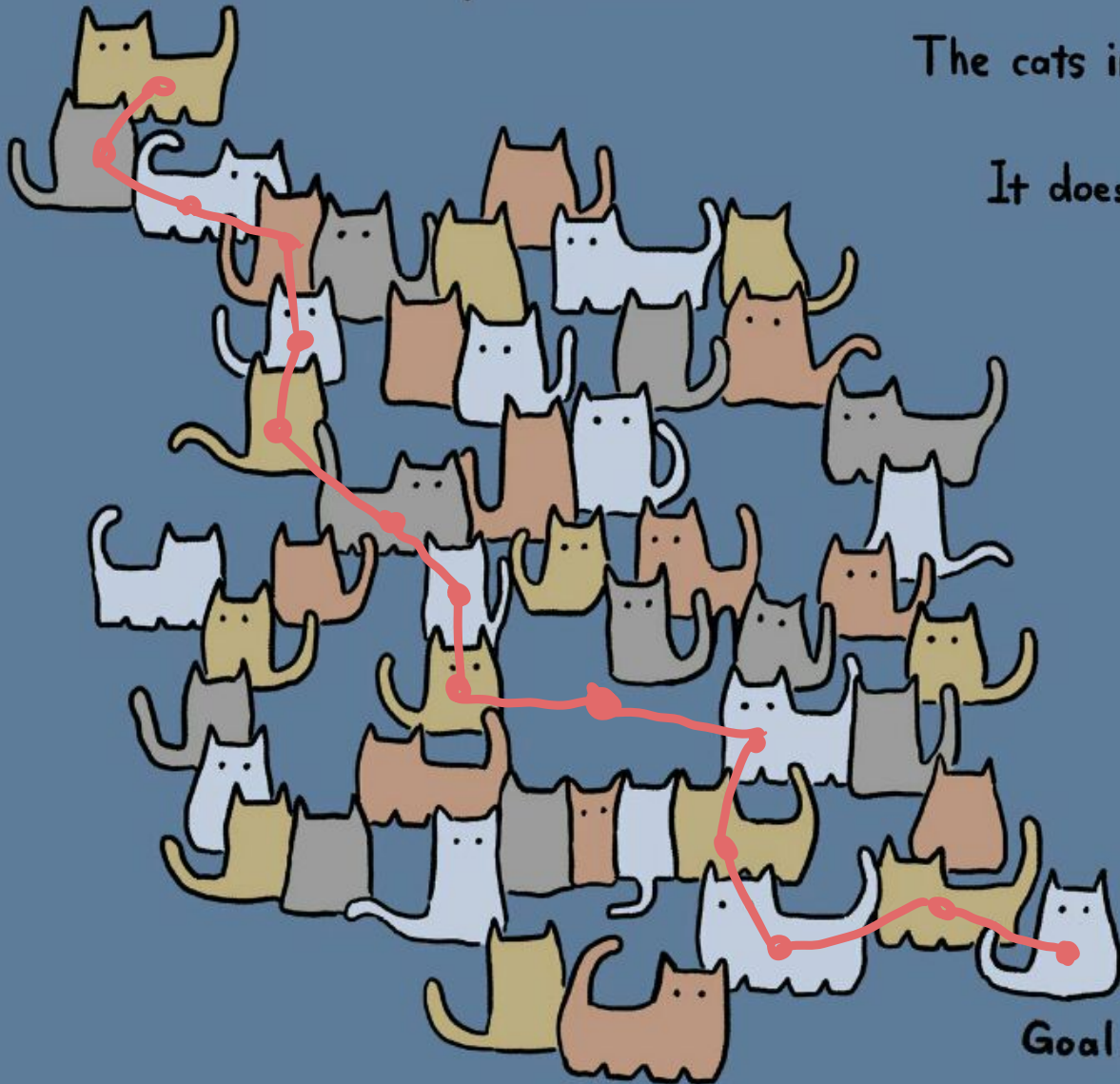
by Tatsunami

Reach the Goal by following the cats, alternating between cats facing the front and cats facing backwards.

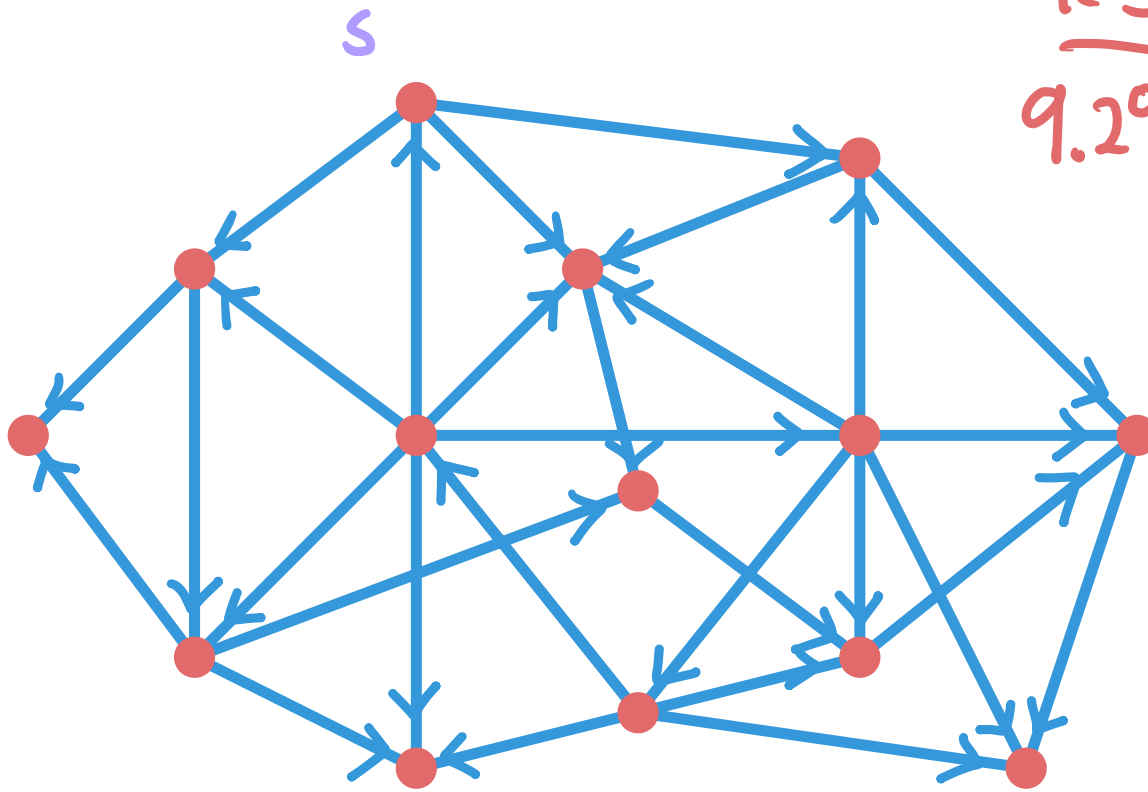
The cats in your path must be connected or overlapping.

It doesn't matter what color or size they are.

Start

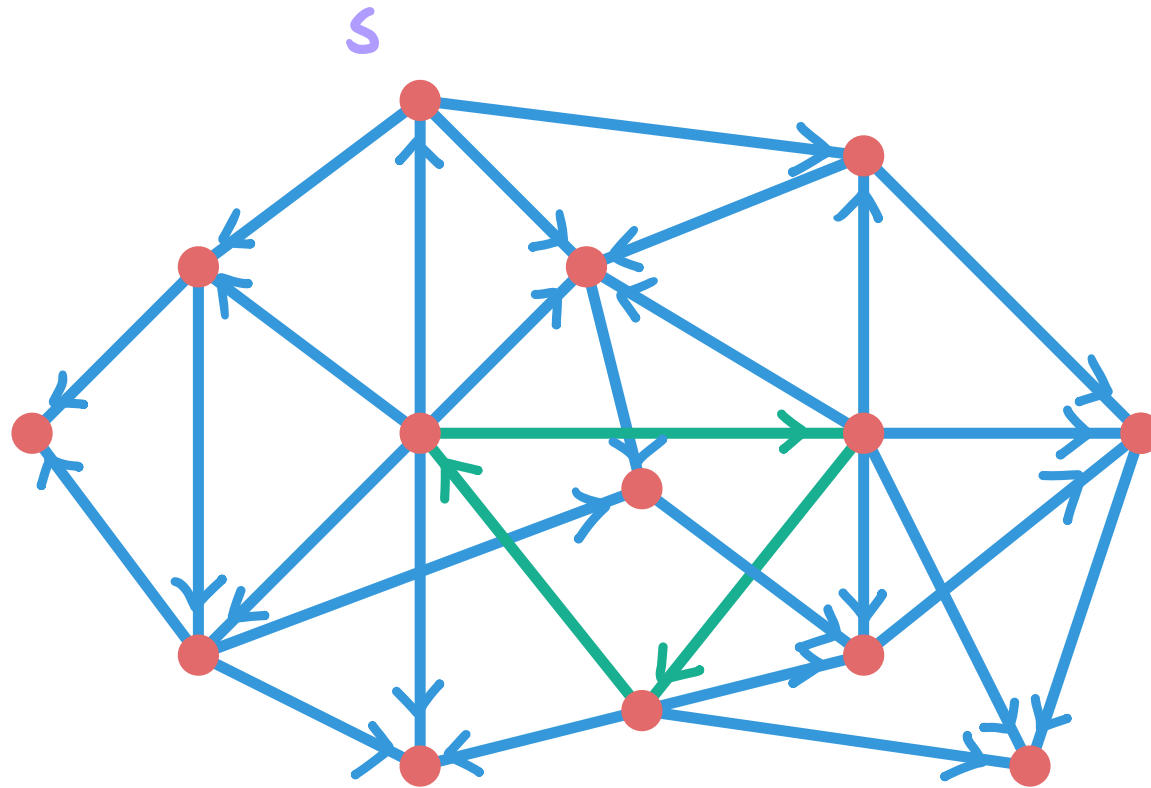


Goal



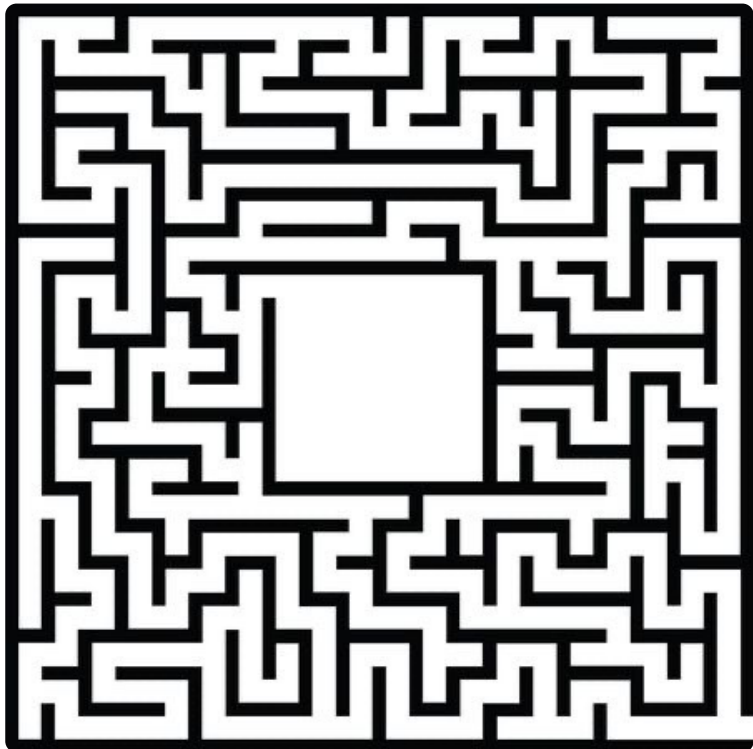
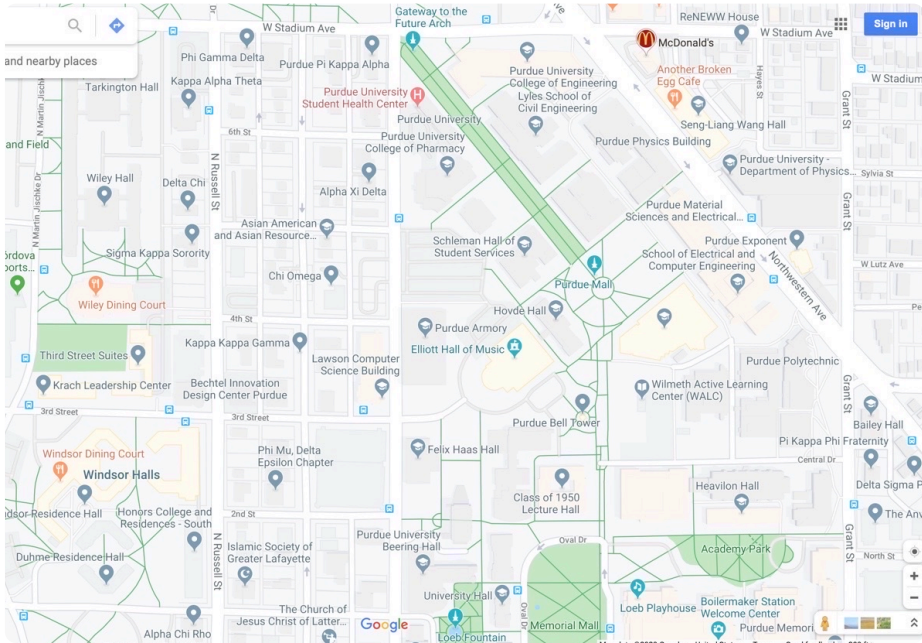
<u>Yes</u>	<u>No</u>	<u>Not vote</u>
9.2%	91.8%	1
	-1	

can s reach all the vertices?



can s reach all the vertices? (no)

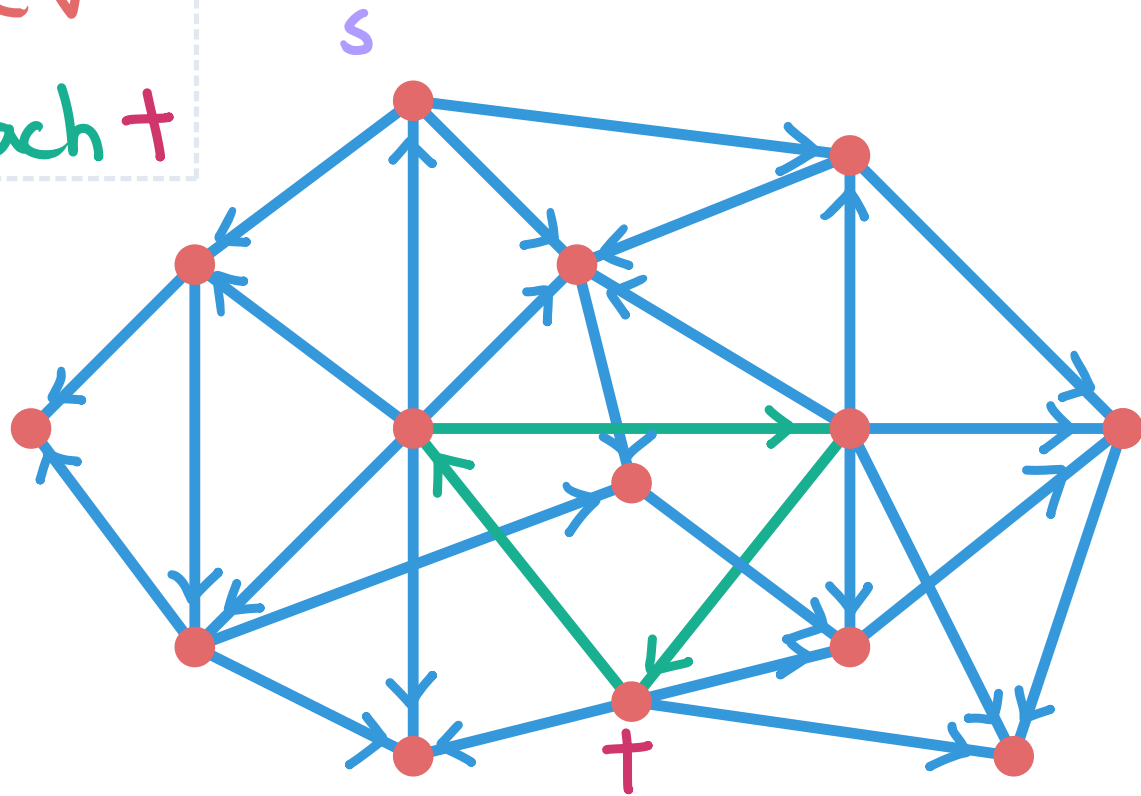
Goal: design an algorithm to figure this out automatically



Input: $G=(V,E)$, $s, t \in V$

Goal: decide if s can reach t

recursive-search(s, t) =
True if s can reach t
False otherwise



Recursive-search(s, t)

1. if $s == t$ return true

2. else

a. $\forall$ out edge (s, v)

if recursive-search(v, t) then return true

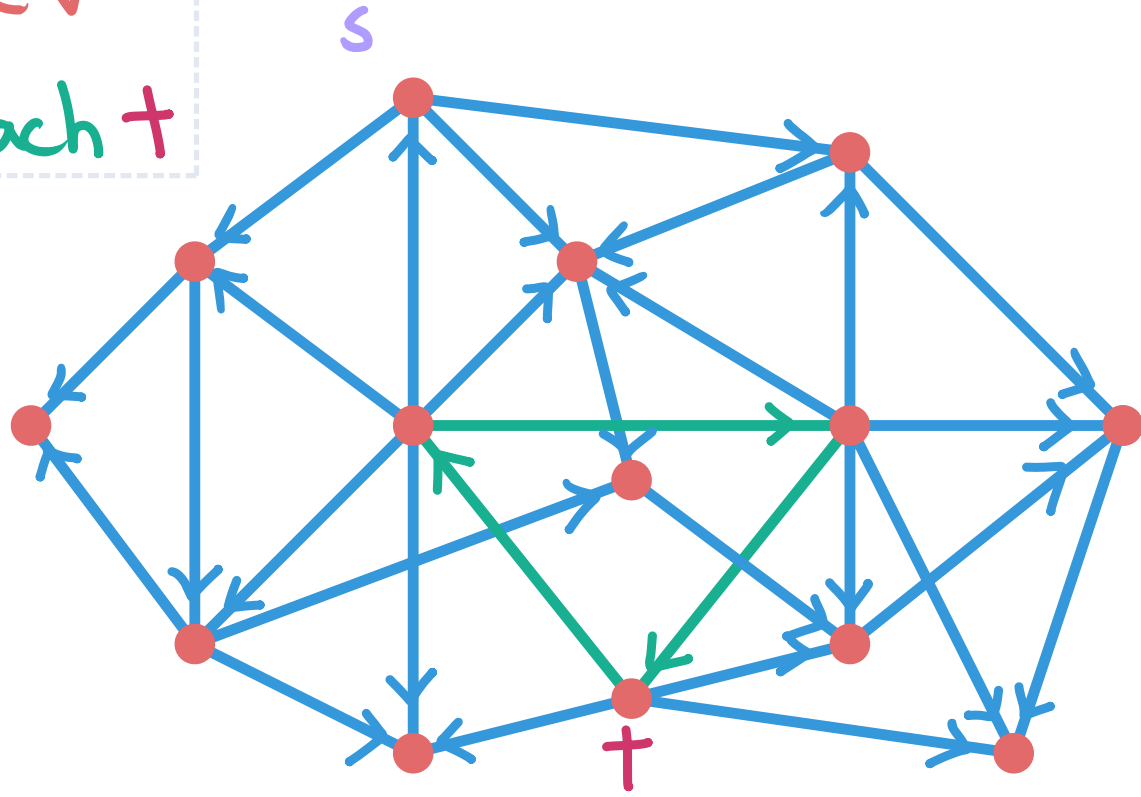
3. return false



Input: $G=(V,E)$, $s, t \in V$

Goal: decide if s can reach t

$\text{recursive-search}(s,t) =$
TRUE if s can reach t ,
and FALSE otherwise



$\text{Recursive-search}(s,t)$

1. if $s=t$ then return TRUE

2. for each edge (s,v)

(a) if $\text{recursive-search}(v,t) = \text{TRUE}$

then return TRUE

3. return FALSE

$\text{recursive-search}(s,t) = \text{TRUE}$ if s can reach t ,
and FALSE otherwise

$\text{recursive-search}(s,t)$

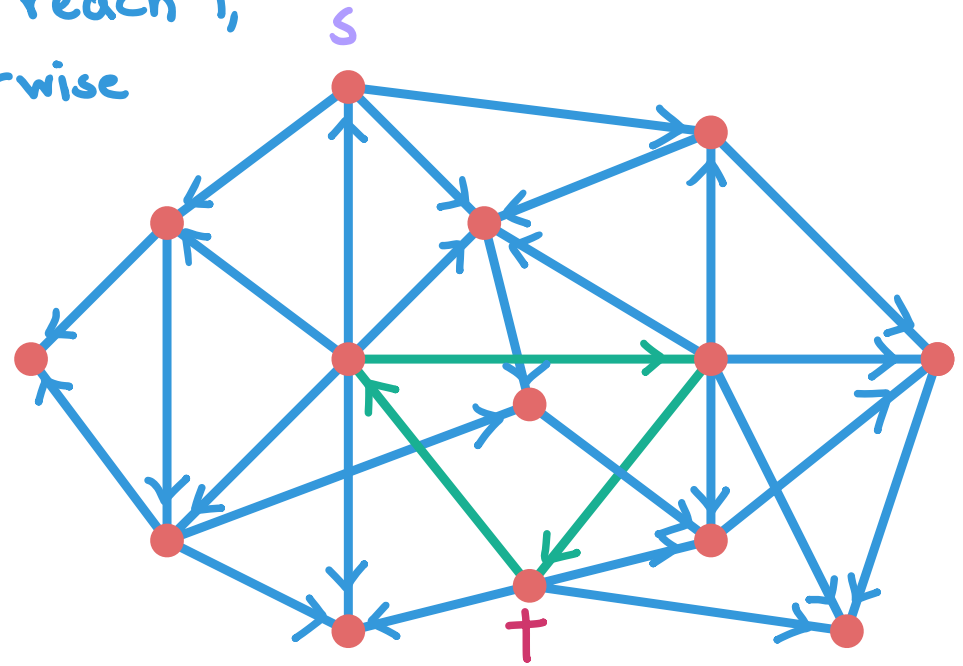
1. if $s=t$ then return TRUE

2. for each edge (s,v)

(a) if $\text{recursive-search}(v,t) = \text{TRUE}$

then return TRUE

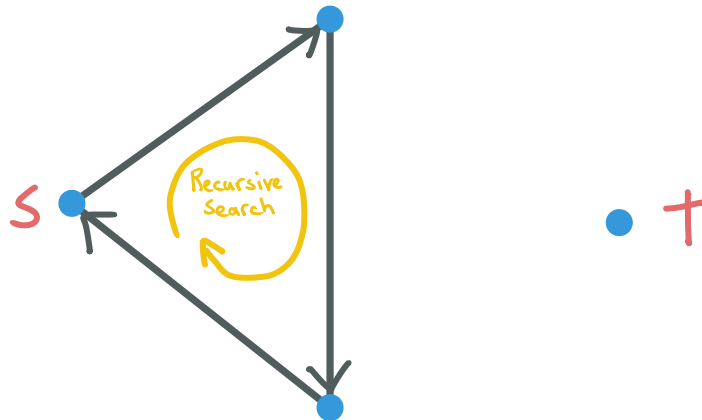
3. return FALSE



Argument: if s can reach t , and $s \neq t$, then there must
be an edge (s,v) s.t. v can reach t .

recursively, $\text{recursive-search}(v,t)$ returns TRUE .

makes sense.
but what about



what went wrong?

Argument: if s can reach t , and $s \neq v$, then there must be an edge (s, v) s.t. v can reach t .

recursively, $\text{recursive-search}(v, t)$ returns TRUE.

recursion requires
induction to be correct

how to fix this?

$\text{recursive-search}(s, t) = \text{TRUE}$ if s can reach t ,
and FALSE otherwise

$\text{recursive-search}(s, t)$

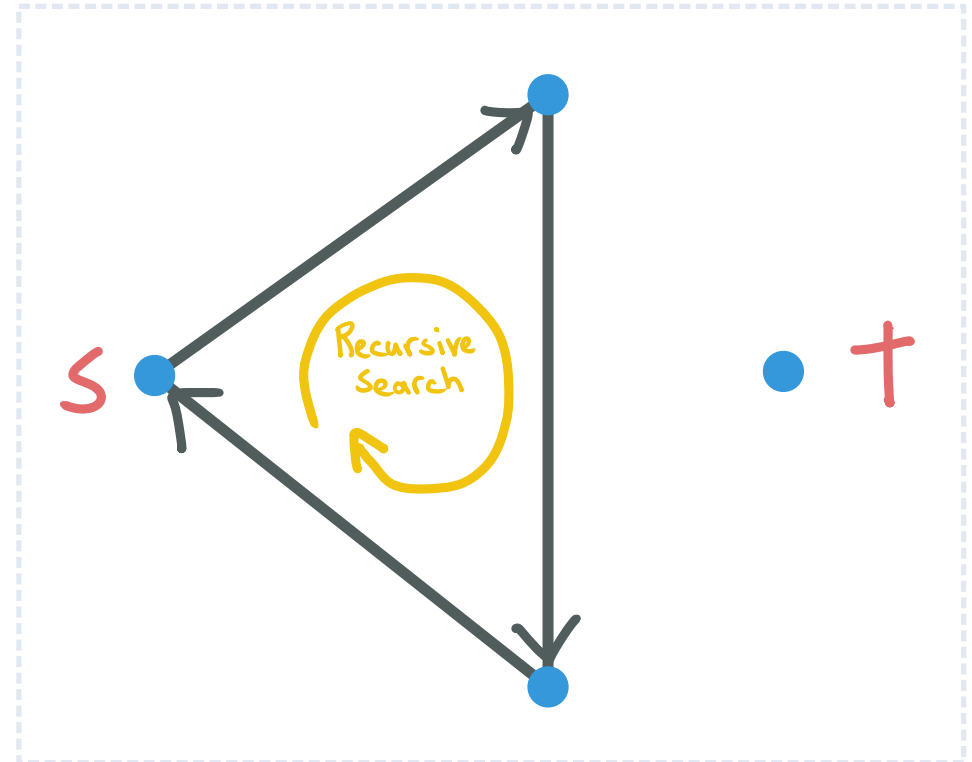
1. if $s = t$ then return TRUE

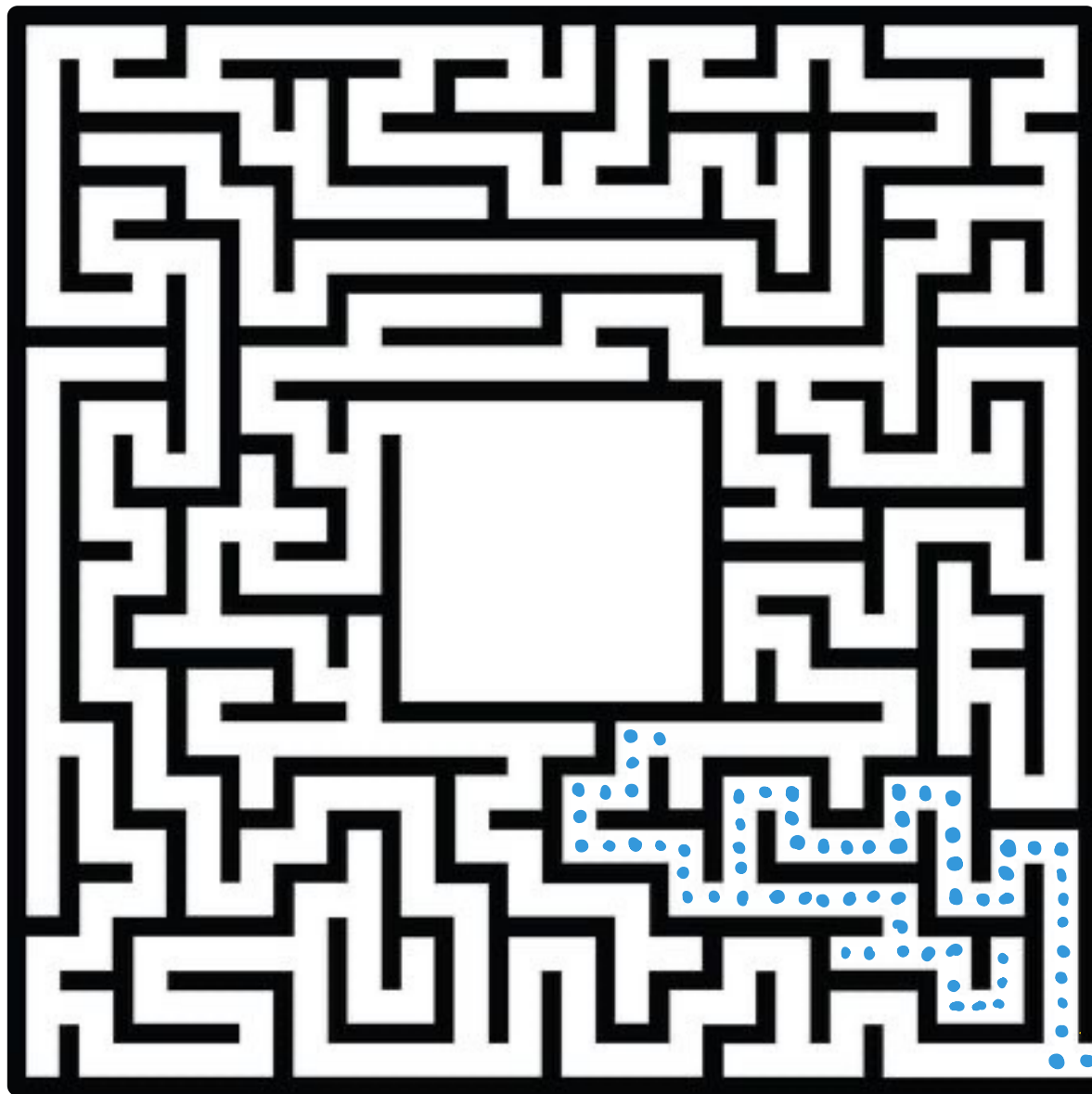
2. for each edge (s, v)

(a) if $\text{recursive-search}(v, t) = \text{TRUE}$

then return TRUE

3. return FALSE





Breadcrumbs

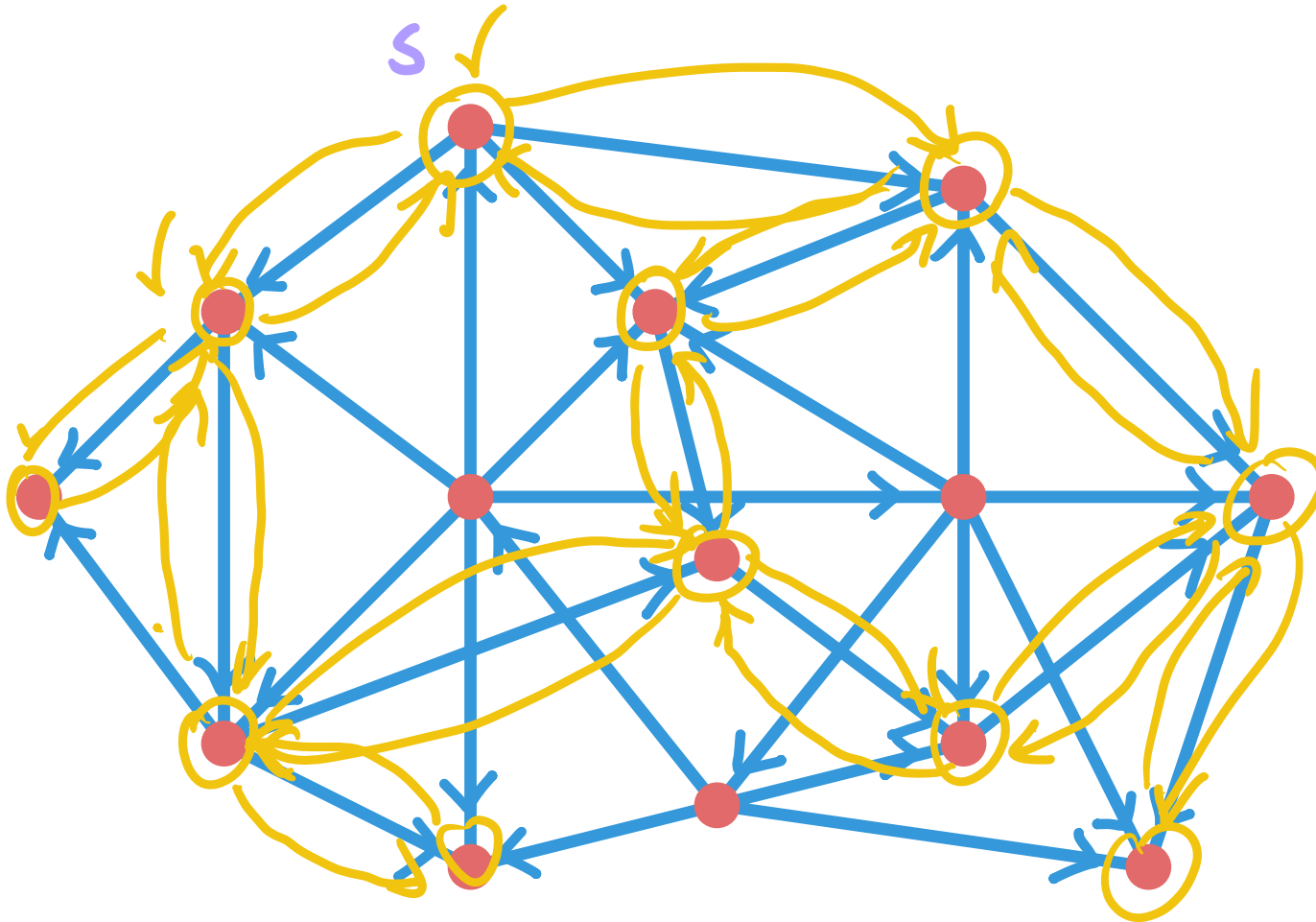
DFS(v)

1. if v is unmarked

A. mark v

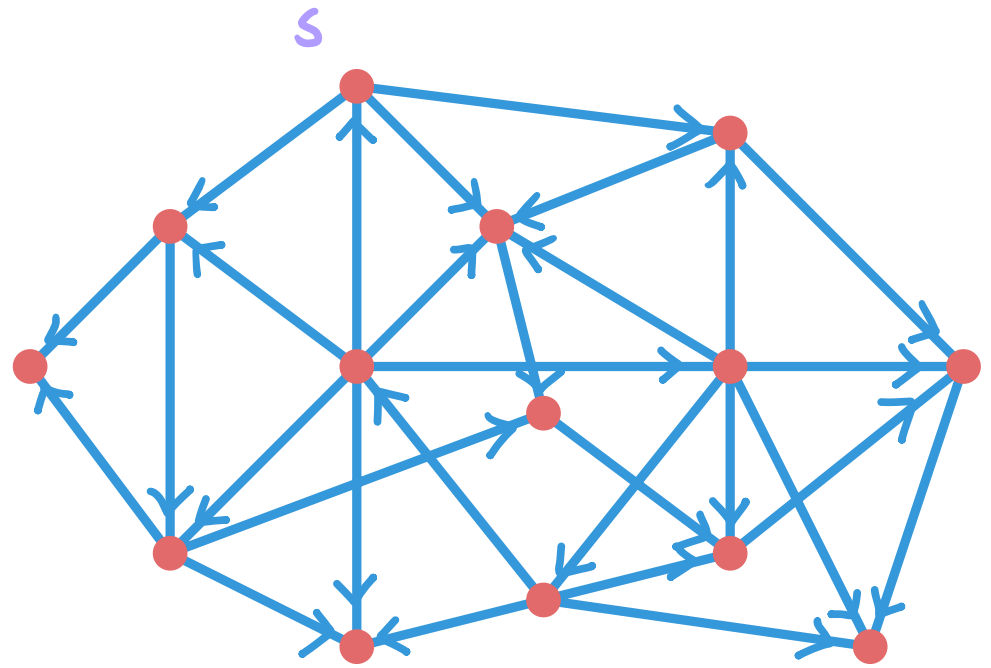
B. for each edge (v, w) from v

1. DFS(w)



DFS(v)

1. if v is unmarked
 - A. mark v
 - B. for each edge (v, w) from v
 1. DFS(w)



2 claims

① DFS(v) runs in $O(m+n)$ time

② DFS(v) visits all vertices reachable from v

DFS(v)

① DFS(v) runs in $O(m+n)$ time

1. if v is unmarked

A. mark v

B. for each edge (v, w) from v

1. DFS(w)

Proof.

DFS(v)

① DFS(v) runs in $O(m+n)$ time

1. if v is unmarked

A. mark v

B. for each edge (v, w) from v

1. DFS(w)

Proof.

we traverse edge (u, v) only after marking u

we mark each vertex once

$\Rightarrow$ traverse each edge once

DFS(v): visit all vertices reachable from

v w/out revisiting v
unmarked

$v \rightarrow w \rightarrow t$

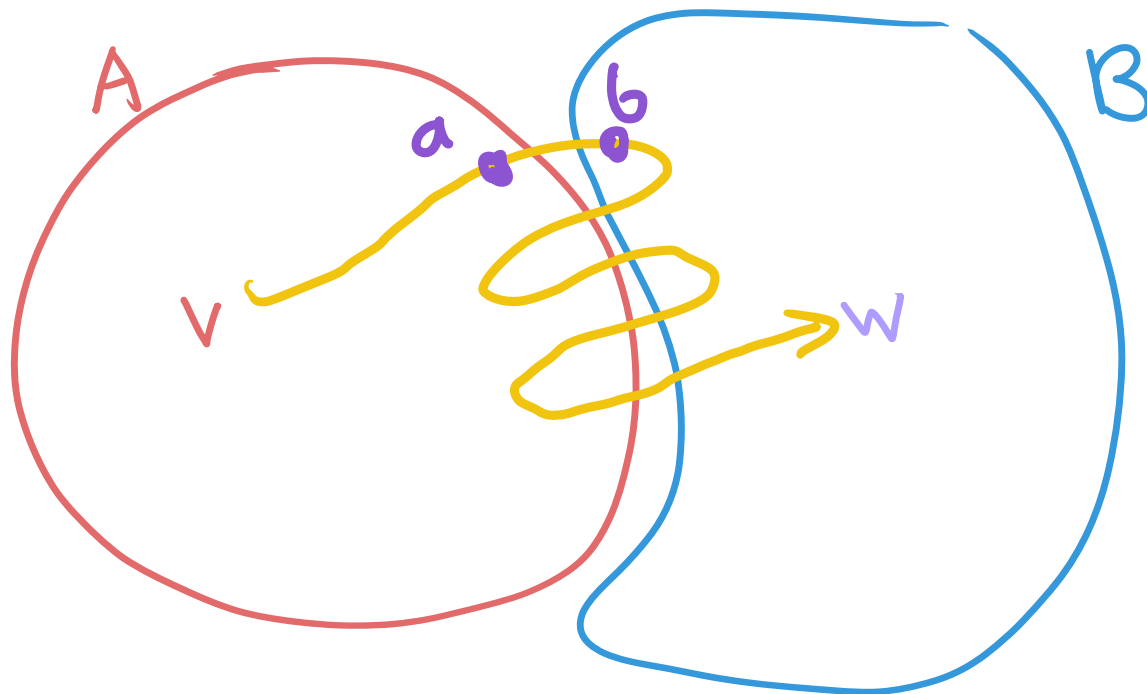
② DFS(v) visits all vertices reachable from v

$A = \{\text{vertices } a \text{ that are marked}\}$

$B = V \setminus A = \{\text{unmarked vertices}\}$

let w be reachable from v

suppose by contradiction that $w \in B$



DFS(v)

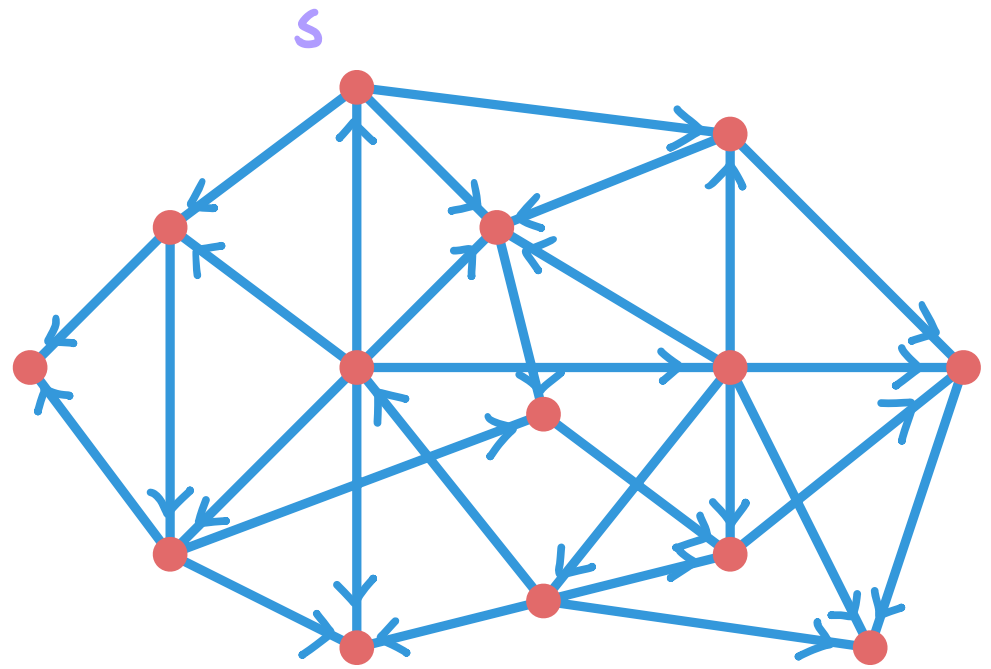
1. if v is unmarked
 - A. mark v
 - B. for each edge (v, w) from v
 1. DFS(w)

consider any path from v to w

since the path starts in A and ends in B , there must be some directed edge from some $a \in A$ and some $b \in B$

DFS(v)

1. if v is unmarked
 - A. mark v
 - B. for each edge (v, w) from v
 1. DFS(w)



2 claims

- ① DFS(v) runs in $O(m+|n|)$ time
- ② DFS(v) visits all vertices reachable from v

more precisely: DFS(v) returns only after all vertices reachable from v have been marked

2 claims

① DFS(v) runs in $O(m+n)$ time

② DFS(v) visits all vertices reachable from v

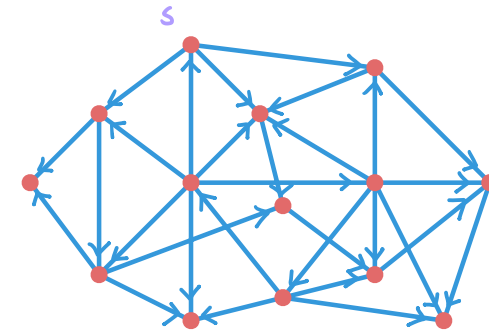
DFS(v)

1. if v is unmarked

A. mark v

B. for each edge (v, w) from v

1. DFS(w)



I. DAG's and topological orderings

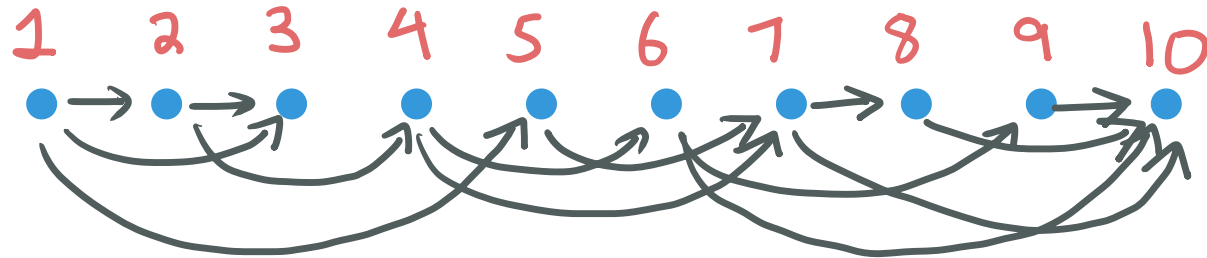
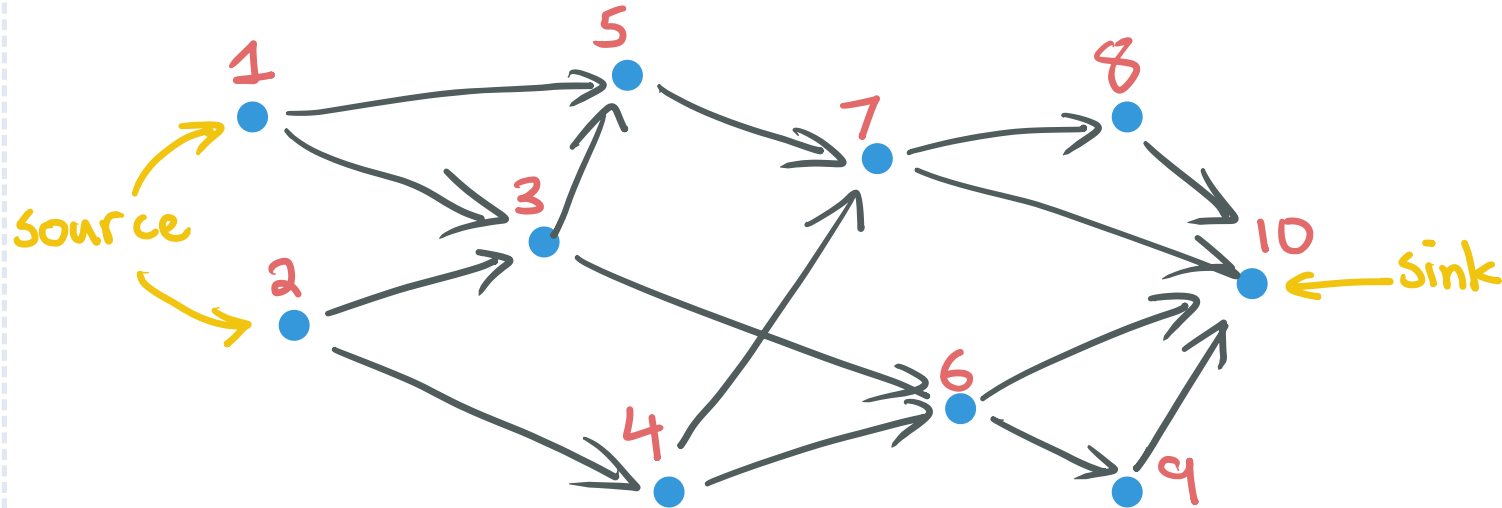
II. Searching graphs

III. Sorting graphs

Recall:

How to topologically sort:
repeatedly remove sources

Topological ordering (of V):
any ordering consistent w/ edges

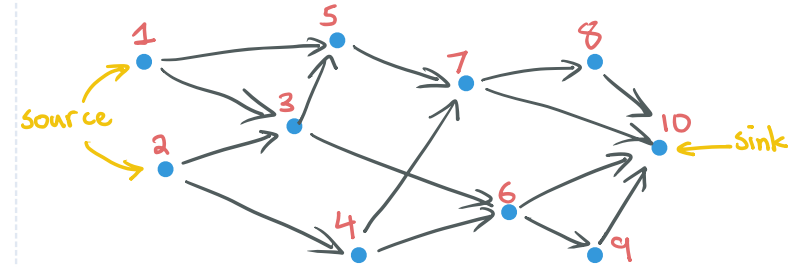


DFS(v)

1. if v is unmarked
 - A. mark v
 - B. for each edge (v, w) from v
 1. DFS(w)

How to topologically sort:
repeatedly remove sources

Topological ordering (of V):
any ordering consistent w/ edges



between the time that

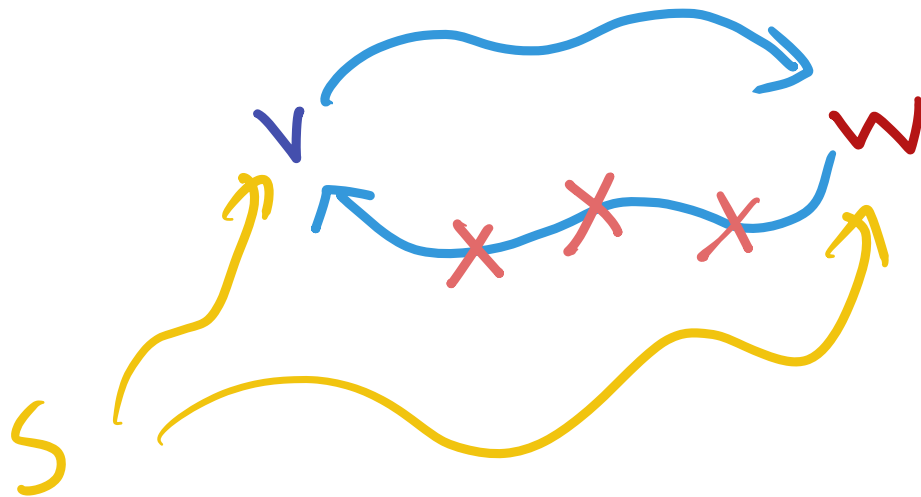
v is marked and DFS(v) exits

DFS only searches vertices reachable from v

Lemma let $v, w \in V$ where

- v can reach w
- w cannot reach v .

Then any DFS that explores both v & w must finish $\text{DFS}(w)$ before $\text{DFS}(v)$



$\text{DFS}(v)$

1. if v is unmarked

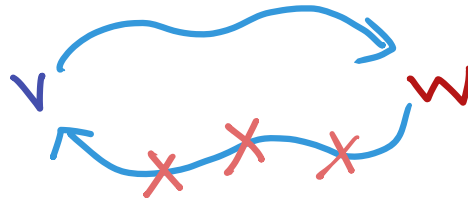
A. mark v

B. for each edge (v, w) from v

1. $\text{DFS}(w)$

Lemma let $v, w \in V$ where

- v can reach w
- w cannot reach v .



DFS(v)

1. if v is unmarked
 - A. mark v
 - B. for each edge (v, w) from v
 1. DFS(w)

Then any DFS that explores both v & w must finish DFS(w) before DFS(v)

Proof

Case 1: DFS marks v first

then DFS marks w and

finishes DFS(w) before finishing DFS(v)

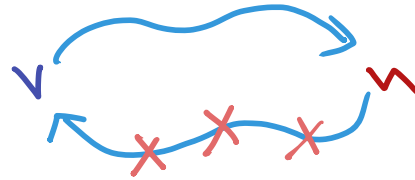
Case 2: DFS marks w first

the DFS(w) only searches vertices reachable

from w — which excludes v — before finishing

Lemma let $v, w \in V$ where

- v can reach w
- w cannot reach v .



Then any DFS that explores both v & w must finish $\text{DFS}(w)$ before $\text{DFS}(v)$

post-DFS order: print out vertices v as we finish $\text{DFS}(v)$

post-DFS(v)

1. If v is unmarked.
 - A. Mark v .
 - B. For each edge (v, w) from v :
 1. post-DFS(w).

/ The following line is the only difference from DFS.*

- C. Append v to the output

post-DFS-driver($G = (V, E)$)

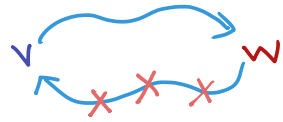
1. For each vertex $v \in V$
 - A. post-DFS(v)



Lemma let $v, w \in V$ where

- v can reach w

- w cannot reach v .



Then any DFS that explores both v & w must finish $\text{DFS}(w)$ before $\text{DFS}(v)$

post-DFS(v)

1. If v is unmarked.

A. Mark v .

B. For each edge (v, w) from v :

1. $\text{post-DFS}(w)$.

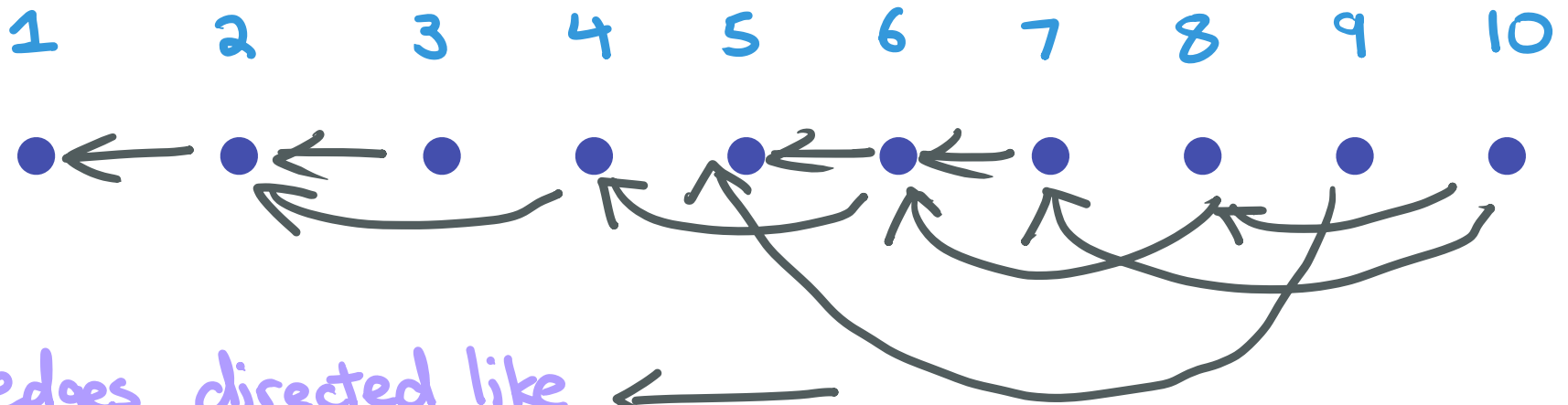
/ The following line is the only difference*

C. Append v to the output

Consider DAG's.

if u can reach v , then v cannot reach u (why?)

so DFS prints v before u



all edges directed like

Reverse topological order

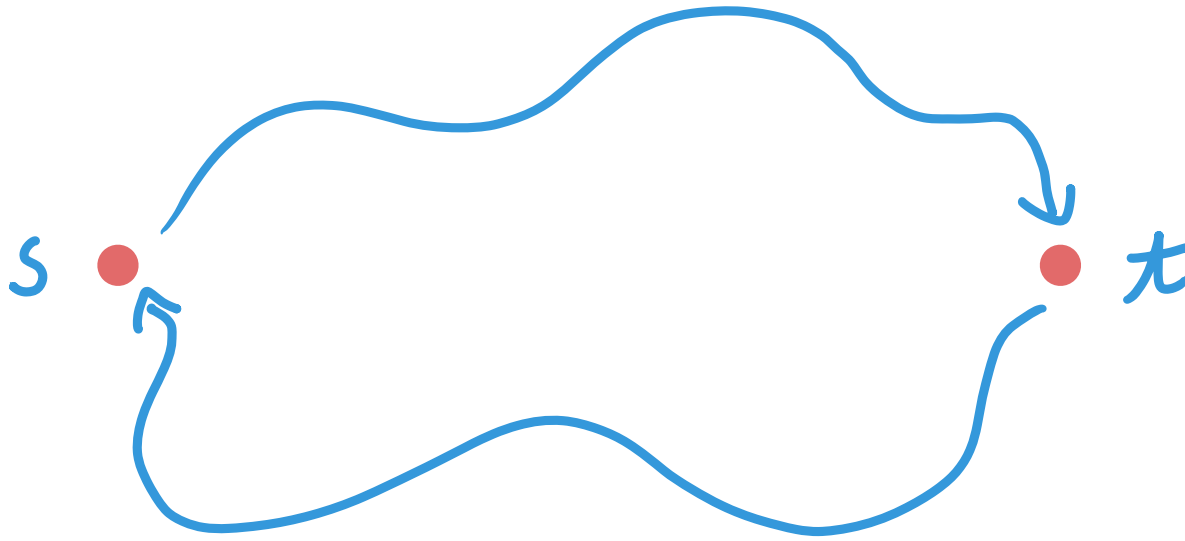
Recall:

Theorem 6.1. *Given a DAG G with m edges and n vertices, one can compute a topological ordering of V in $O(m + n)$ time.*

DFS (+ reversing) gives a second algo.

Strong connectivity

two vertices s and t are strongly connected
if s can reach t and t can reach s

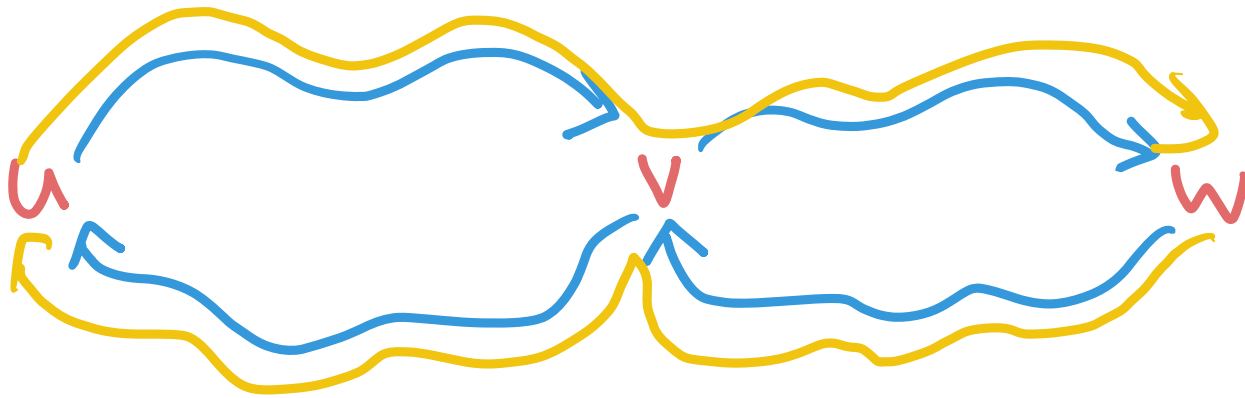


a graph is strongly connected if every pair
of vertices is strongly connected

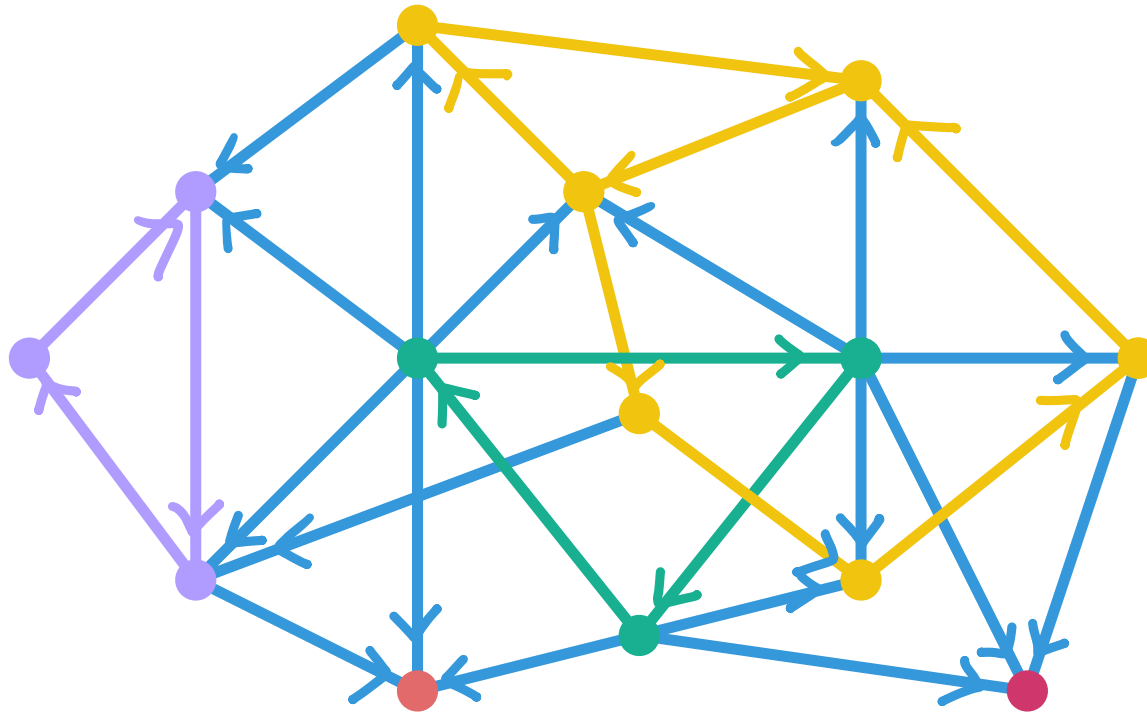
Claim strong connectivity defines an equivalence class

(that is, if u is strongly connected to v , and v is strongly connected to w , then u is strongly connected to w)

Proof



each equivalence class of strongly connected vertices is called a strongly connected component (SCC)

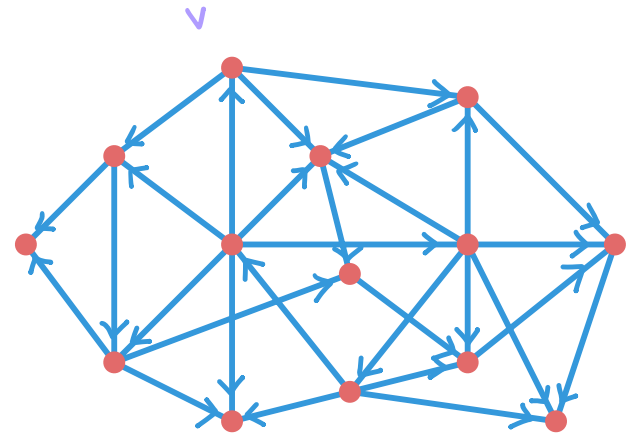


Goal: find the SCC's algorithmically
(strongly connected components)

First:

given v , find the SCC of v

(strongly
connected
component)



ideas?

$\text{dfs}(v)$

run $\text{dfs}(v)$ in reverse

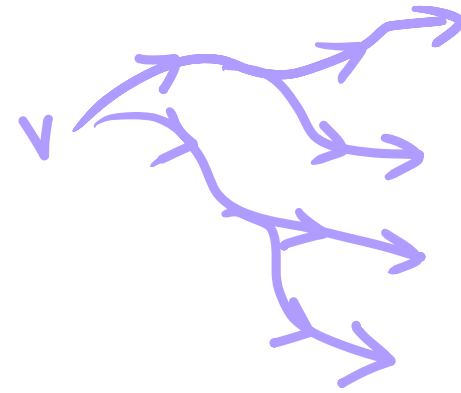
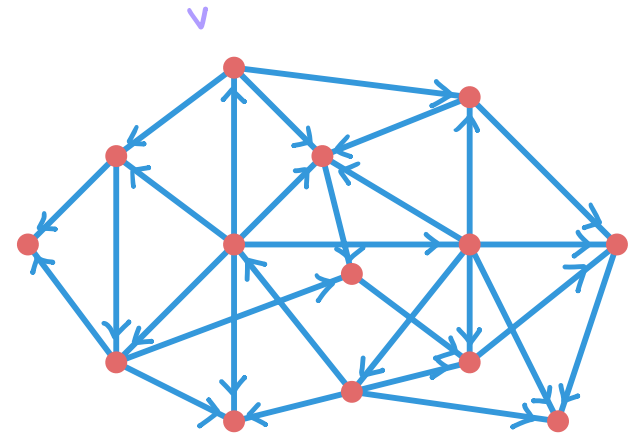
First:

given v , find the SCC of v
(strongly connected component)

① run DFS from v

②*

③



First:

given v , find the SCC of v
(strongly connected component)

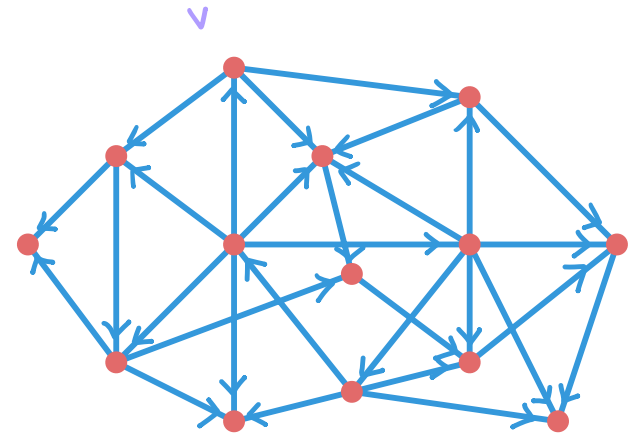
① run DFS from v

②* run DFS from v in the reversed graph

③ return set of vertices visited in both DFS's

*② visits all vertices that can reach v in original graph

don't change the algorithm; change the graph!



SCC(v)

- ① run DFS from v
- ②* run DFS from v in the reversed graph
- ③ return set of vertices visited in both DFS's

to get all the SCC's:

run from every vertex

$$n \times O(m+n) = O(mn)$$

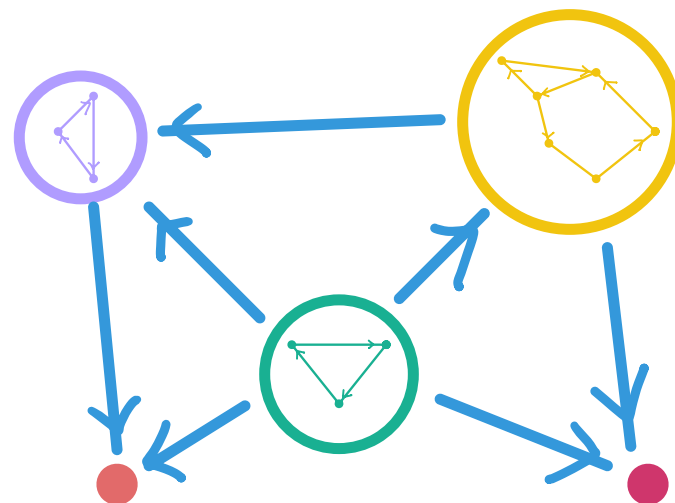
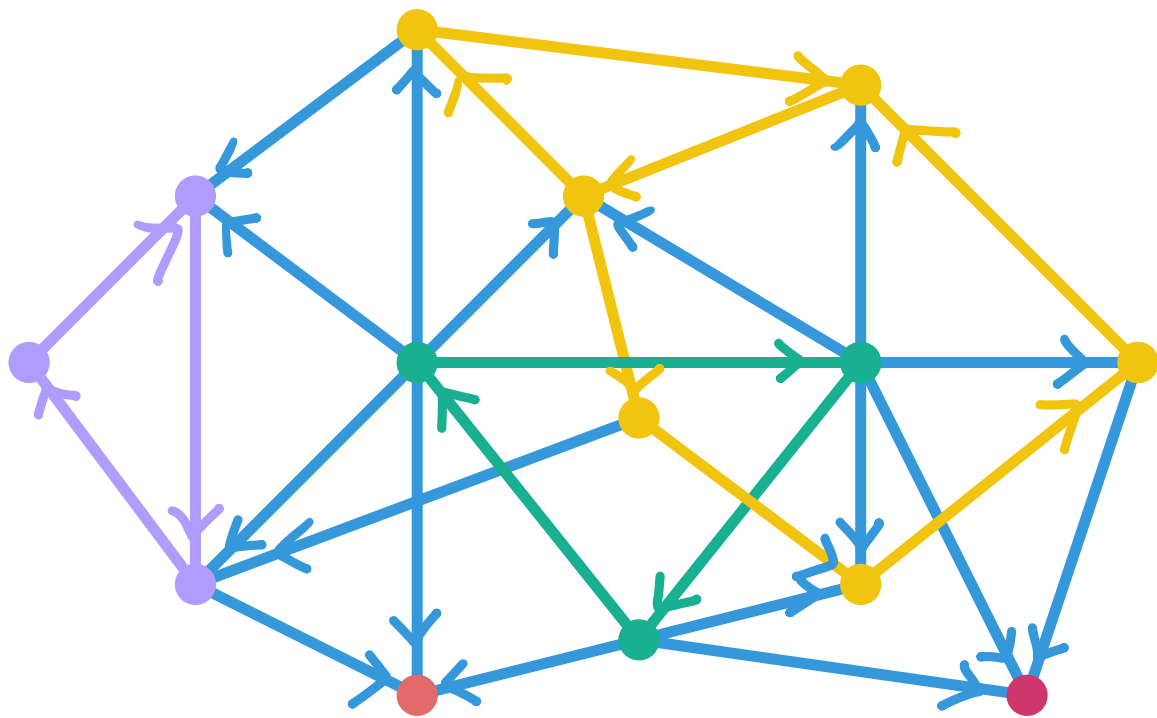
$$O(m+n)$$

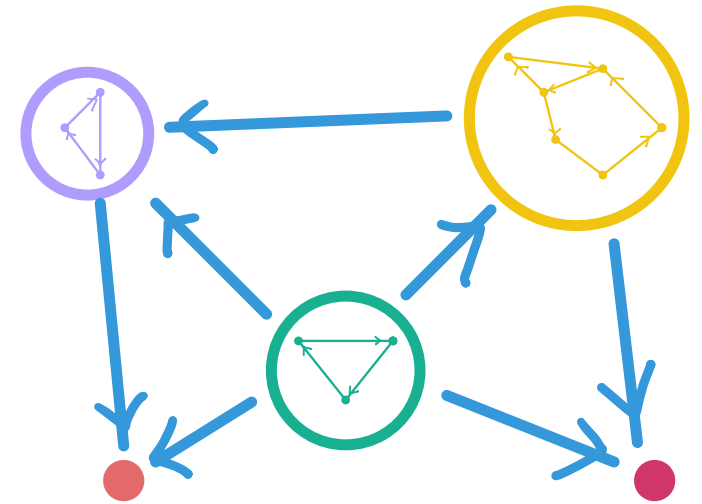
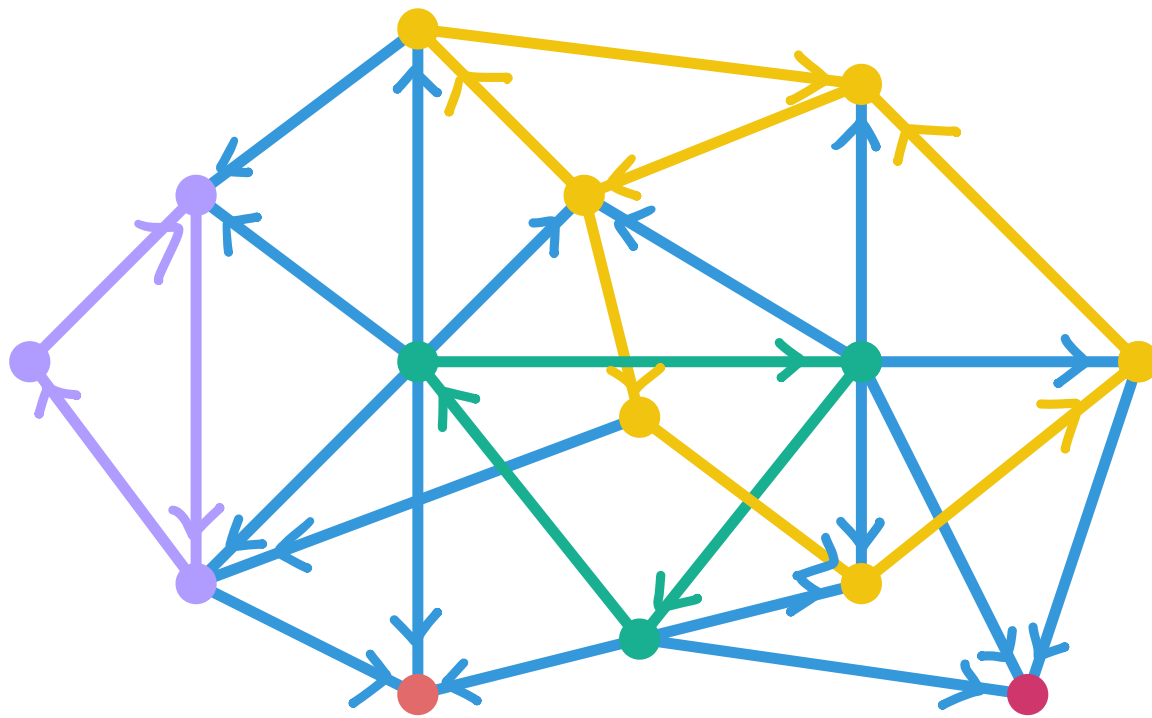
$$O(m+n)$$

$$O(n)$$

$$O(m+n)$$

better?





Conspicuously: no cycles a DAG!
 (directed acyclic graph)

Our current approach

Slow-SCC's ($G=(V,E)$)

1. until $V=\emptyset$

a. let v be any vertex in V

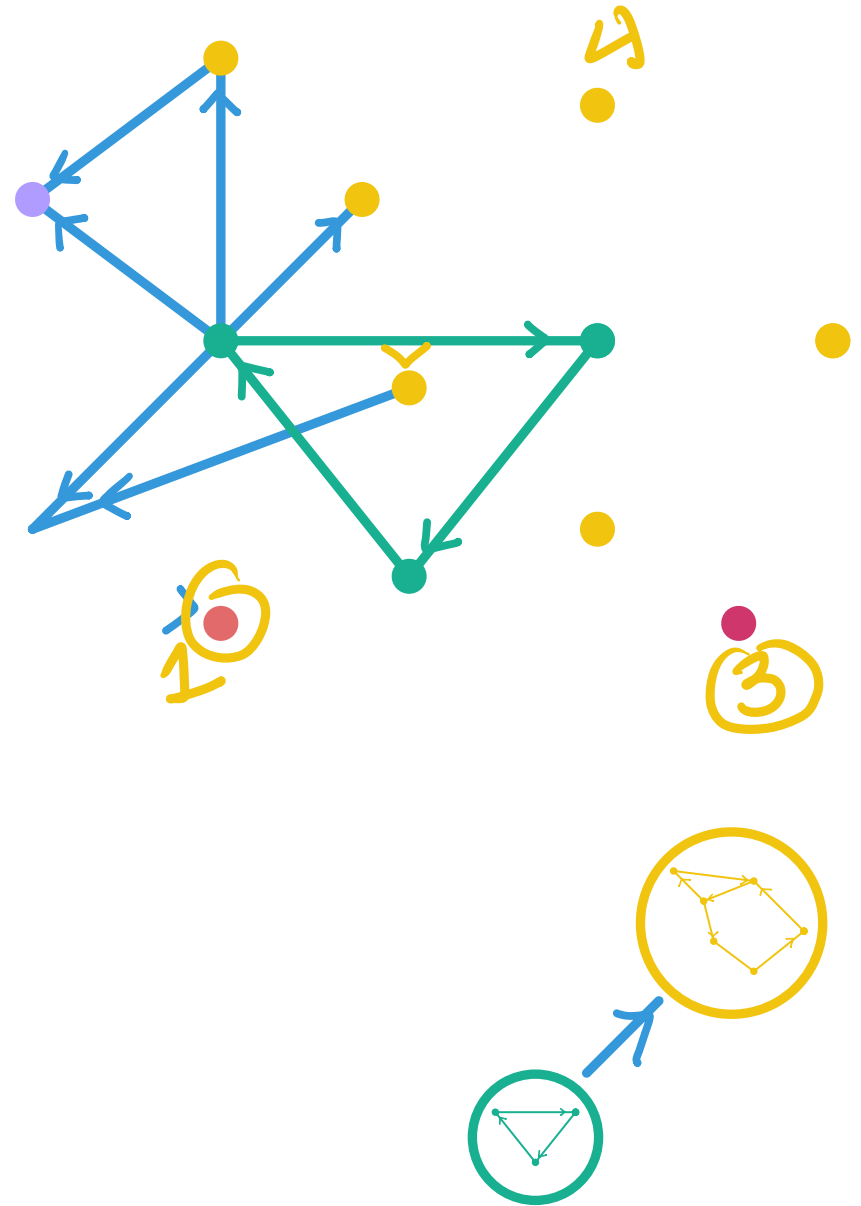
b. $C \leftarrow \text{SCC}(v)$

c. label each vertex in C w/
(some label for C)

d. remove C and all incident
edges from G

(this takes $O(mn)$)

(2)



sink-first

~~slow~~ - SCC's ($G=(V,E)$)

until $V=\emptyset$

let v be any vertex ~~in V~~

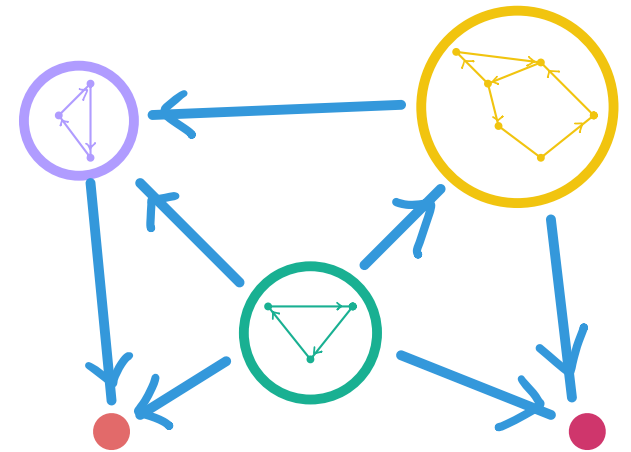
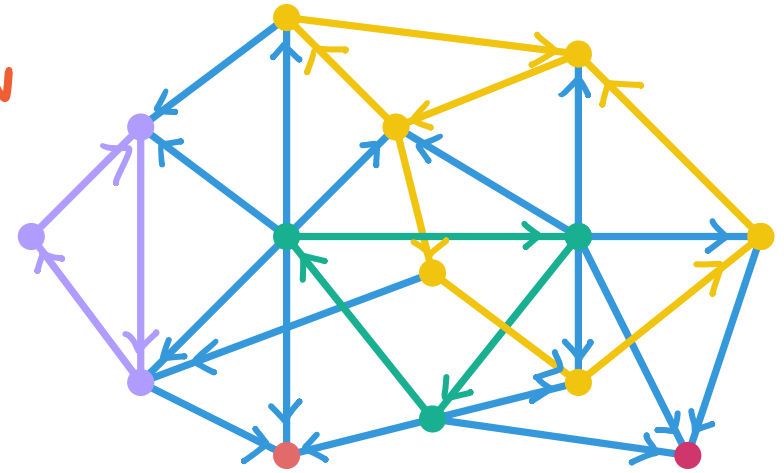
DFS

$C \leftarrow \text{SCC}(v)$ // DFS cannot leave sink component

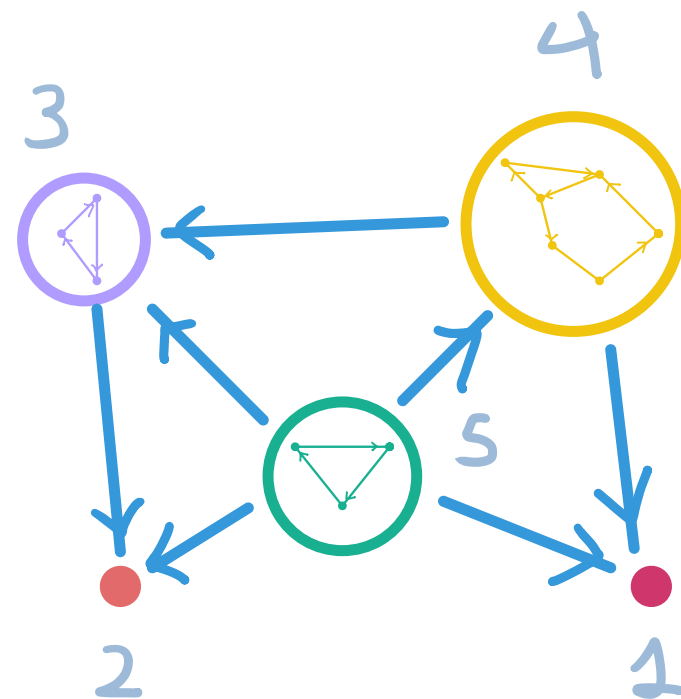
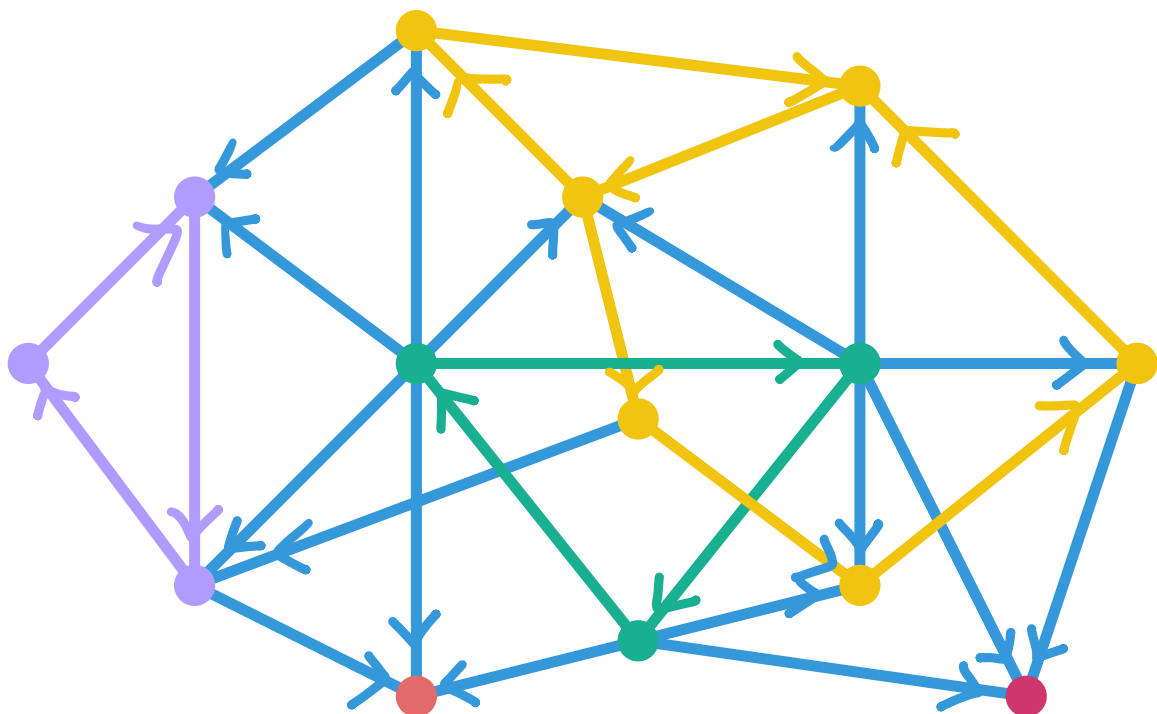
label each vertex in C w/
(some label for) C

remove C and all incident
edges from G

we don't know
how to do this
yet

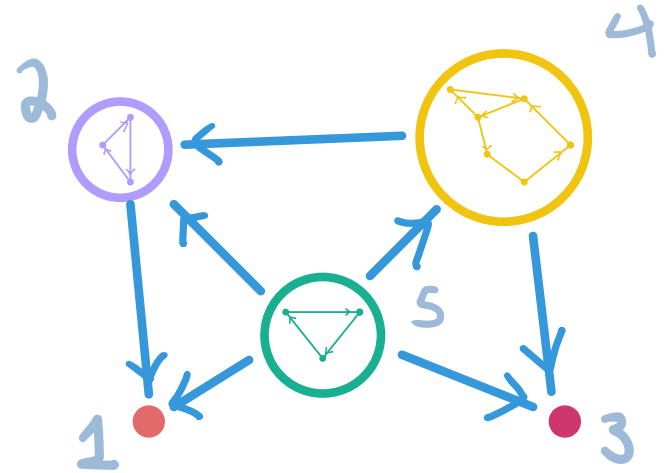
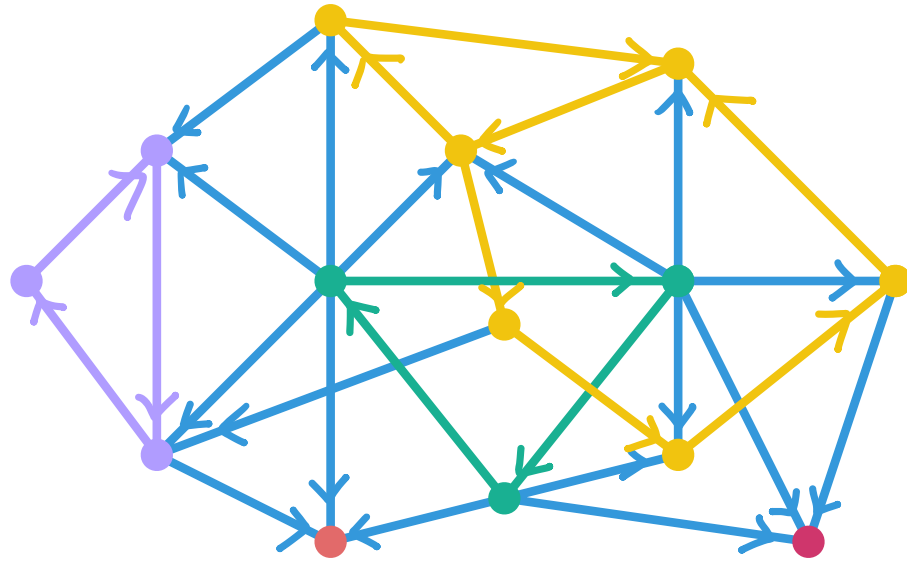


we only visit each vertex once!



To wit:

if we can figure out "sink components" on the fly, then

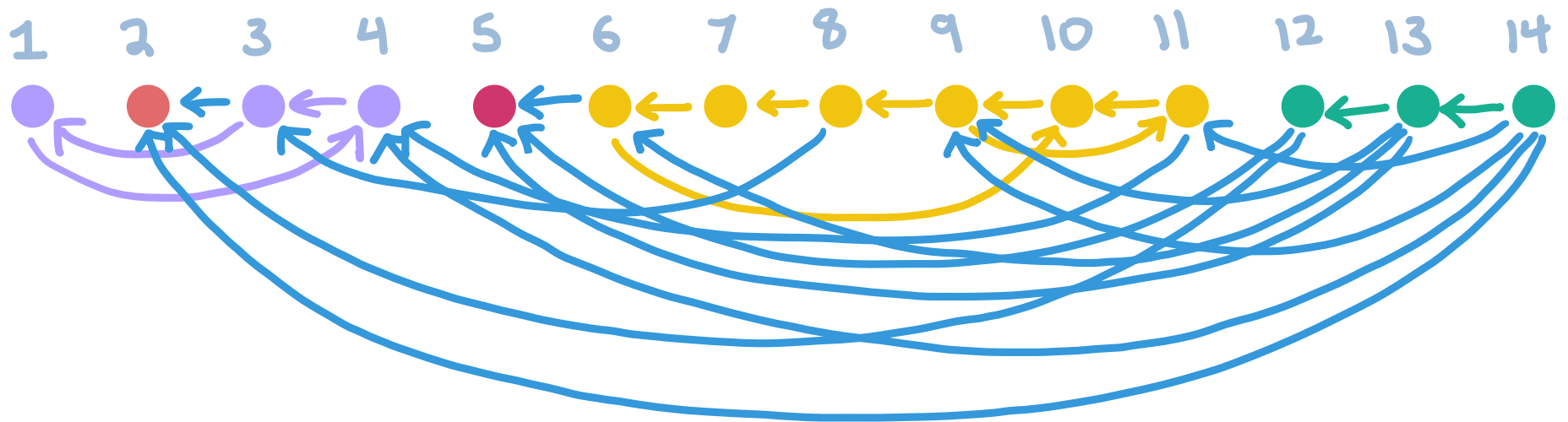
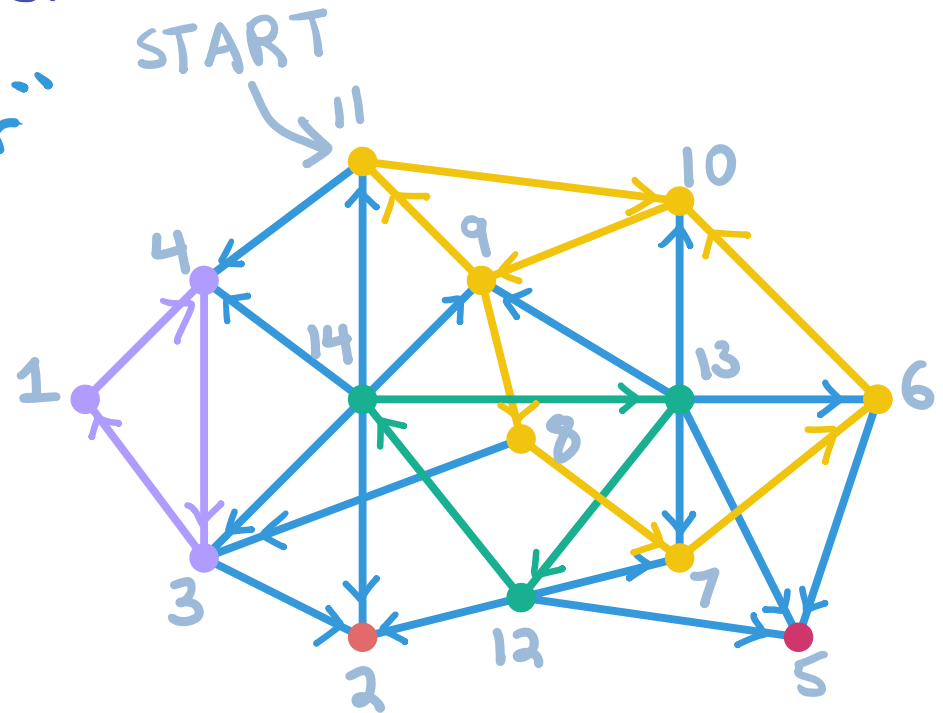


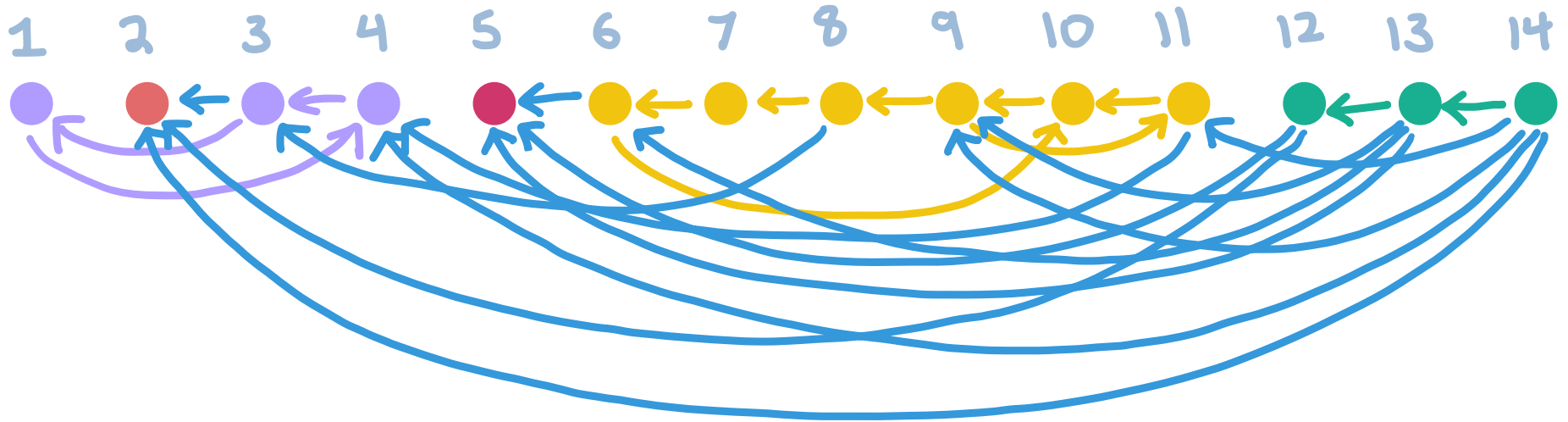
(a) only 1 DFS per v (instead of reversing)

(b) $O(m+n)$ time total:

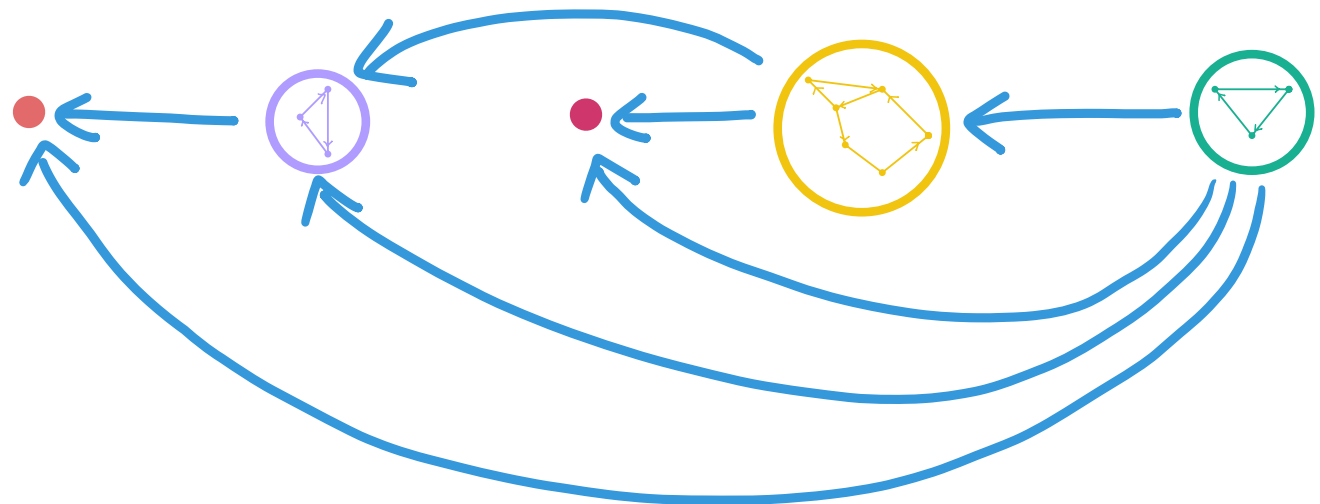
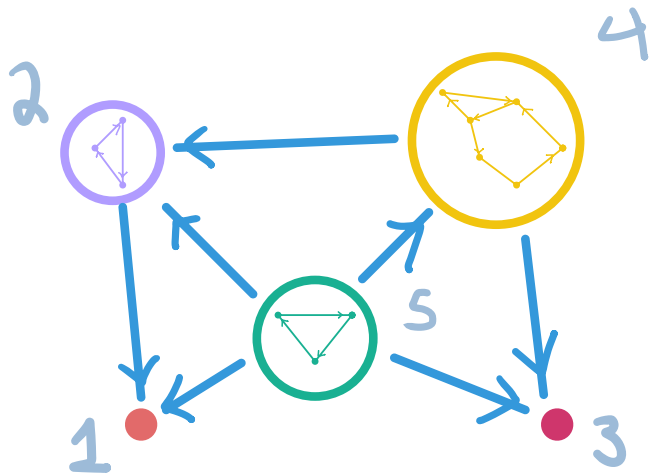
each DFS(v) does not leave v 's ^(strongly connected component) SCC

Recall: post-order DFS order
lists vertices in "sink-first order"
(when G is a DAG)





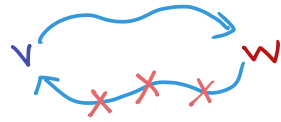
all cross-component edges [in BLUE] are in the same direction



Lemma let $v, w \in V$ where

- v can reach w

- w cannot reach v .



Then any DFS that explores both v & w must finish $\text{DFS}(w)$ before $\text{DFS}(v)$

post-DFS(v)

1. If v is unmarked.

A. Mark v .

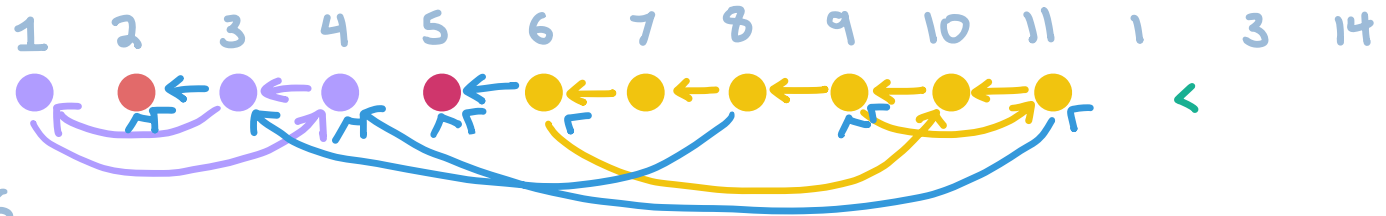
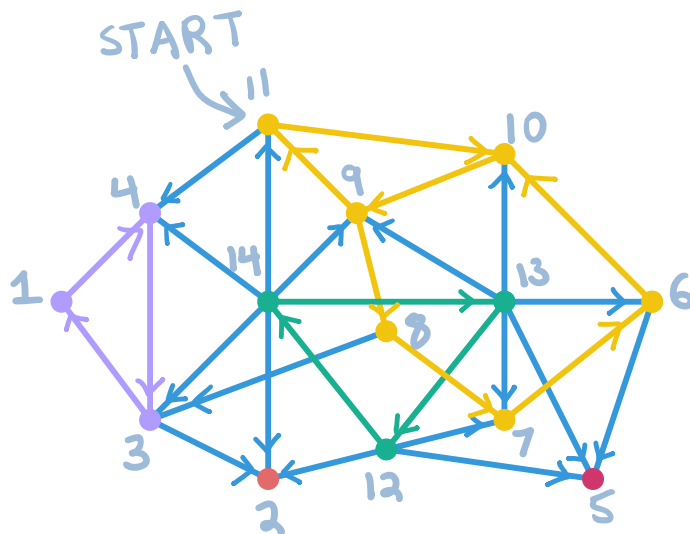
B. For each edge (v, w) from v :

1. $\text{post-DFS}(w)$.

/ The following line is the only difference*

C. Append v to the output

Lemma. let $v_1, \dots, v_n$ list vertices s.t. if v_i can reach v_j , and v_j cannot reach v_i , then $i > j$.
then v_n is from a source component



sink-first

~~slow~~ - SCC's ($G=(V,E)$)

until $V=\emptyset$

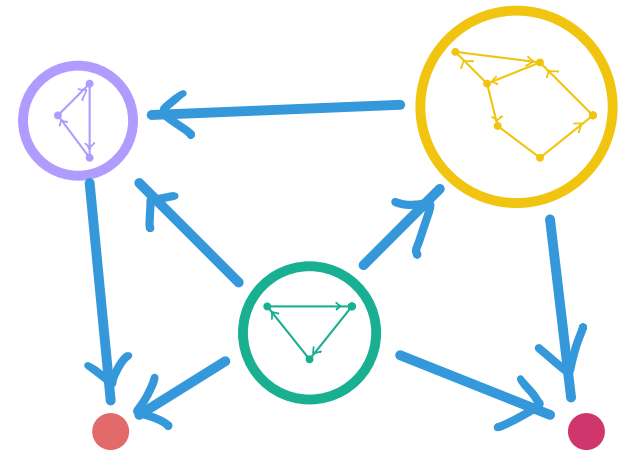
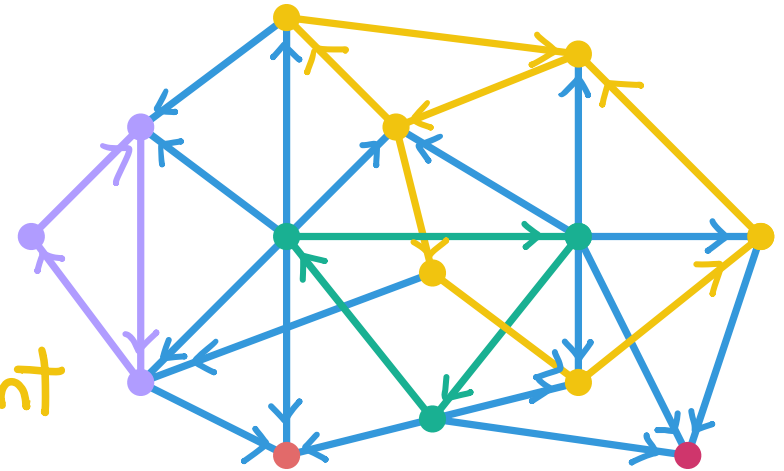
let v be a vertex in a sink component

$C \leftarrow \text{DFS}(v)$ // DFS cannot leave sink component

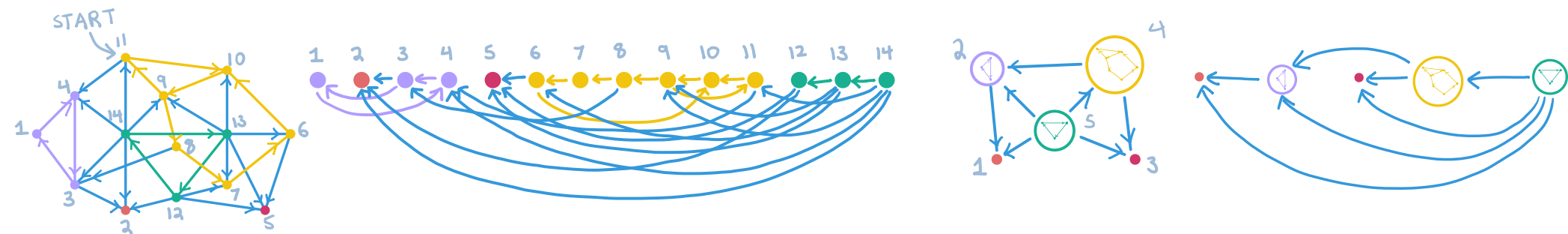
label each vertex in C w/
(some label for) C

remove C and all incident
edges from G

we don't know
how to do this
yet



Lemma. let $v_1, \dots, v_n$ list vertices s.t. if v_i can reach v_j ,
and v_j cannot reach v_i , then $i > j$.
then v_n is from a source component



sink-first-SCC($G = (V, E)$)

1. Compute a post-DFS ordering of V in the *reversed* graph G_r
2. Until $V = \emptyset$
 - A. Let v be the *last* (remaining) vertex in V in the post-DFS order of G_r
// v is in a sink component of G .
 - B. $C \leftarrow$ all vertices marked by $\text{dfs}(v)$
 - C. Append C to the output
 - D. Remove C and all incident edges from G

$O(m+n)$ [!!!]