

Searching & Sorting Graphs

today: directed graphs

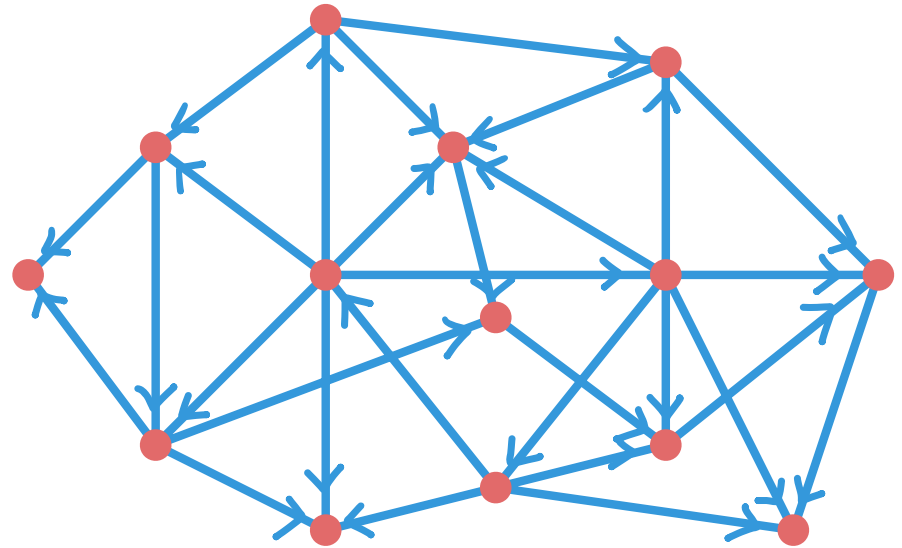
acyclic graphs,

topological sort,

cycles, reachability,

depth-first search,

strongly connected components



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

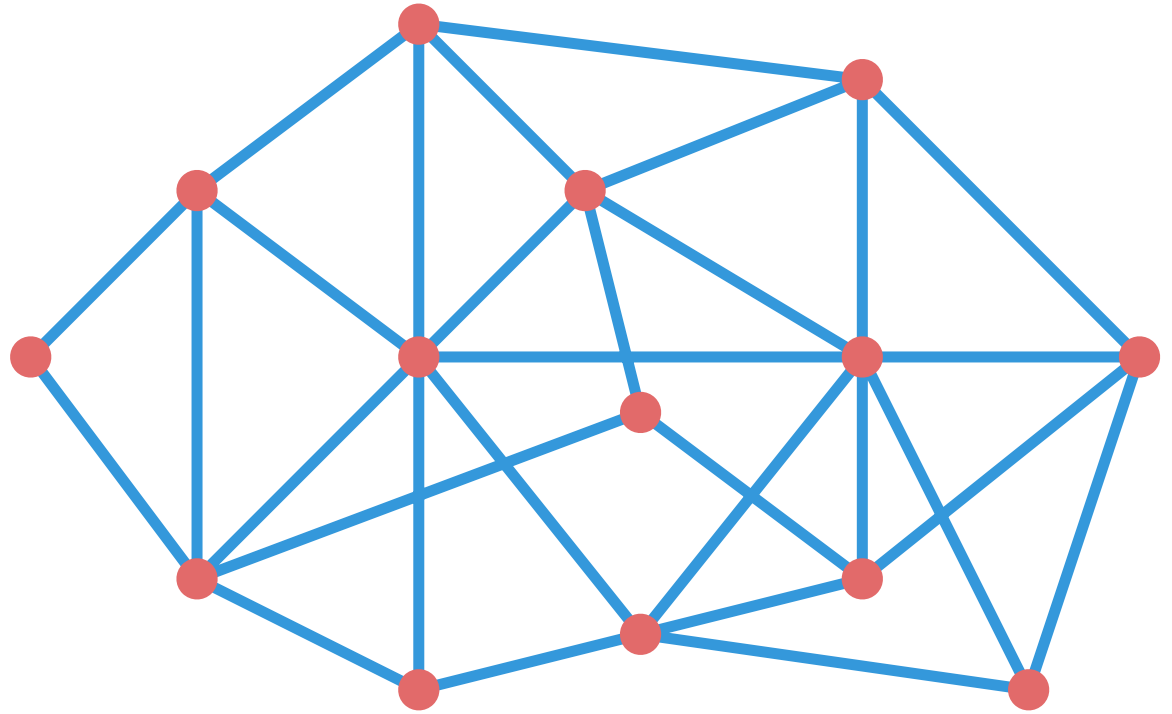
Graphs

$$G=(V, E)$$

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

vertices •

edges —————

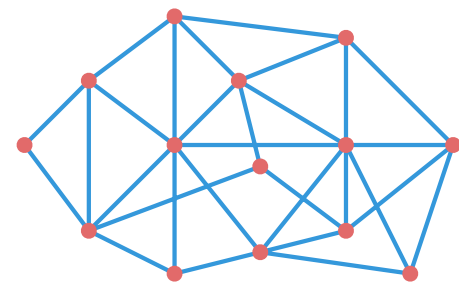


undirected
edge
between u and v



directed
edge
from u to v

Representing graphs



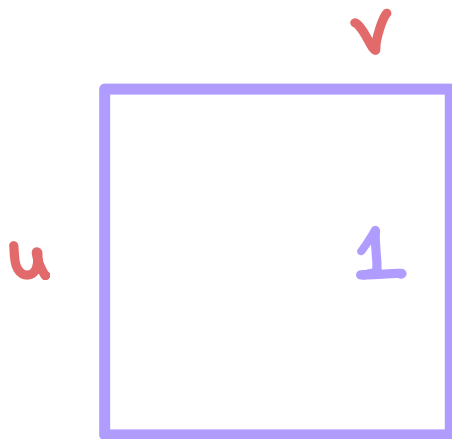
① "Adjacency list"

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)



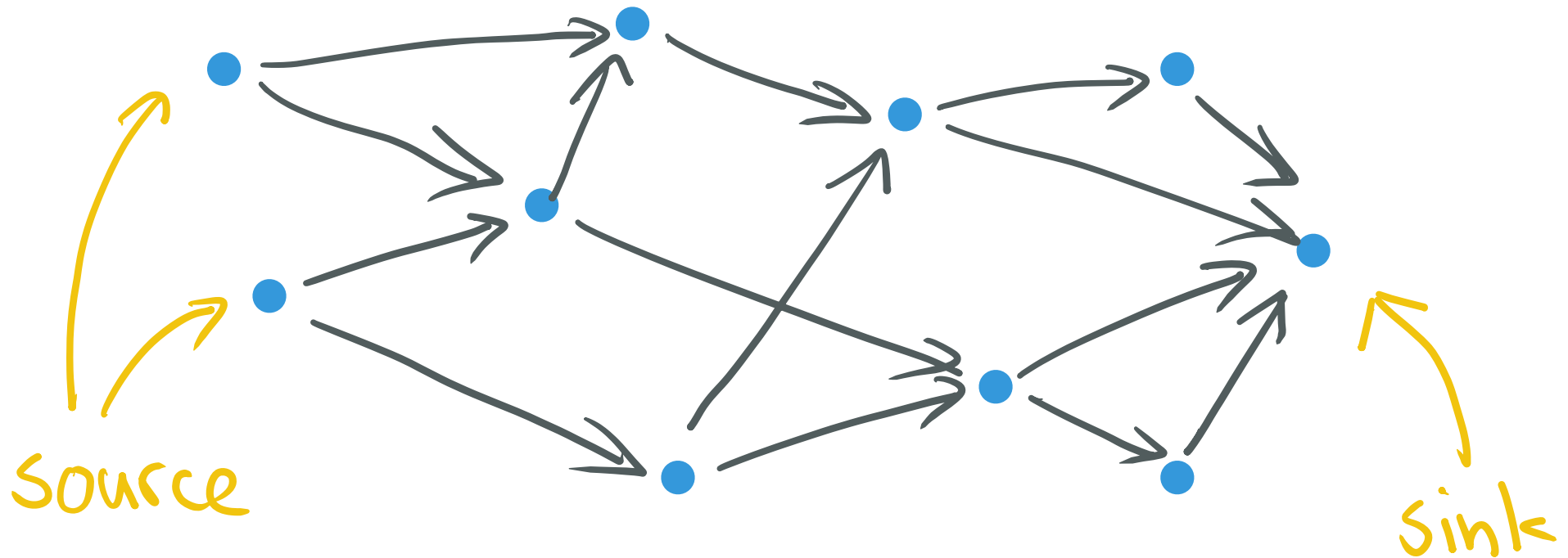
I. DAG's and topological orderings

II. Searching graphs

III. Sorting graphs

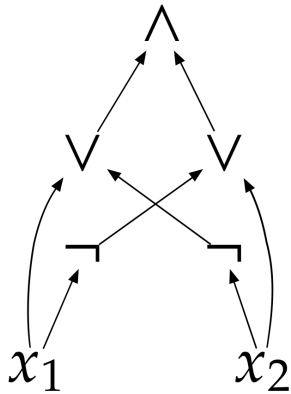
Directed Acyclic Graphs (DAG's)

(no cycles)

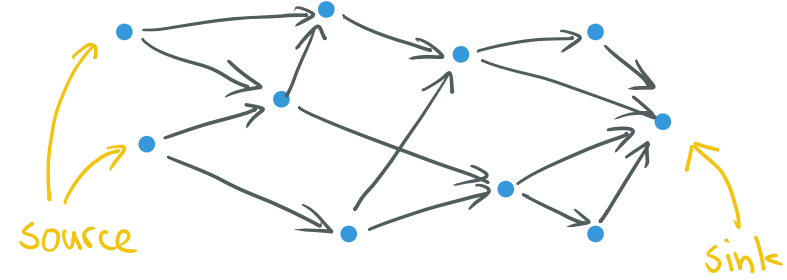


e.g. circuits

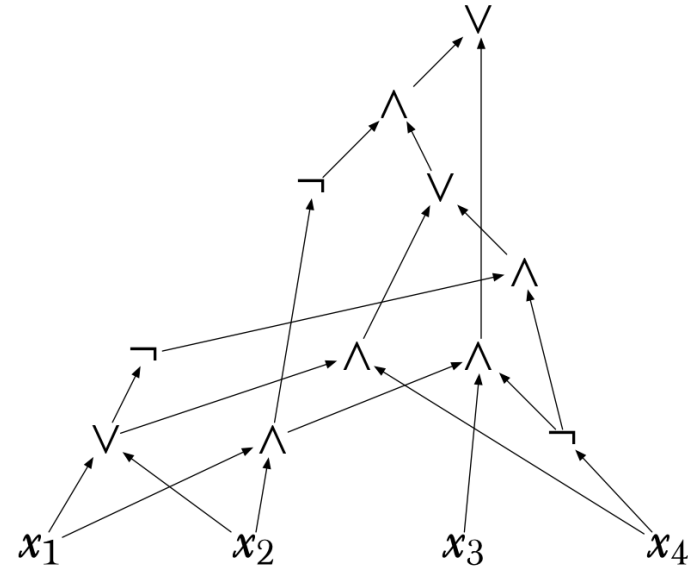
XOR ("exclusive-or")
(TRUE iff $x_1 \neq x_2$)



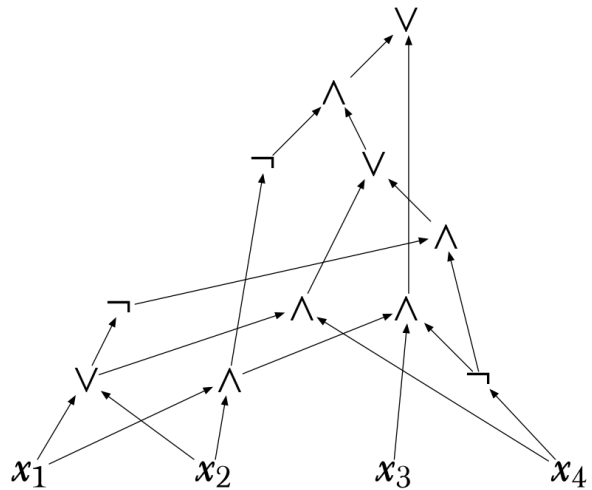
$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$



Addition



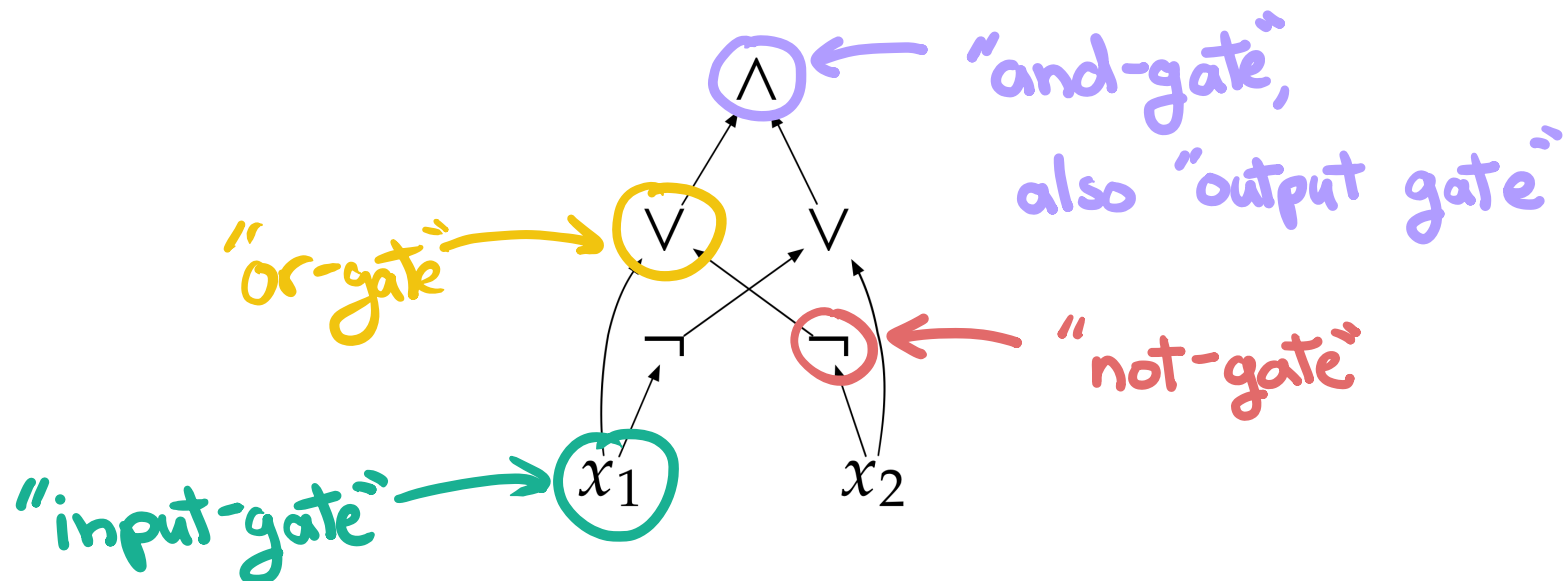
$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{if } x_1 + x_2 = x_3 x_4 \\ 0 & \text{otherwise} \end{cases}$$



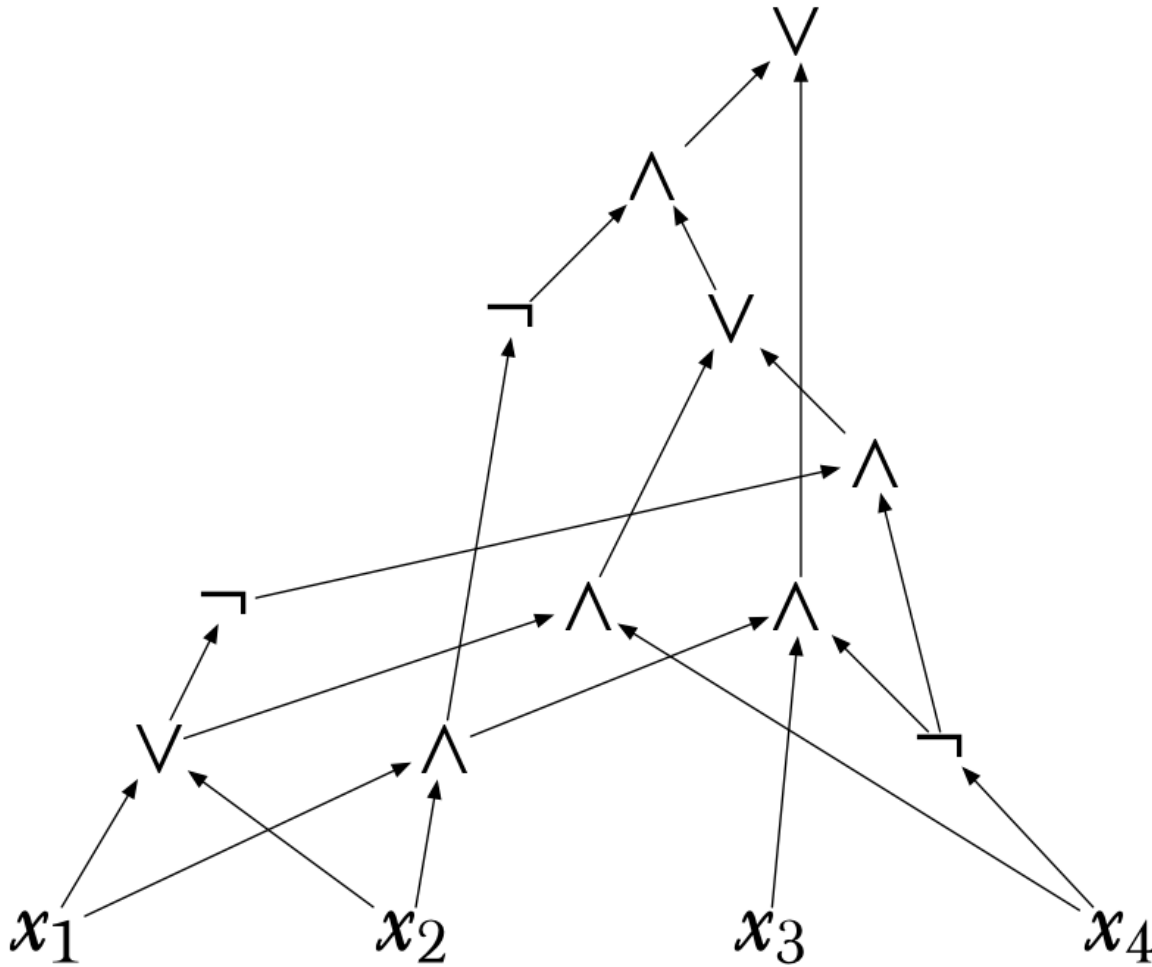
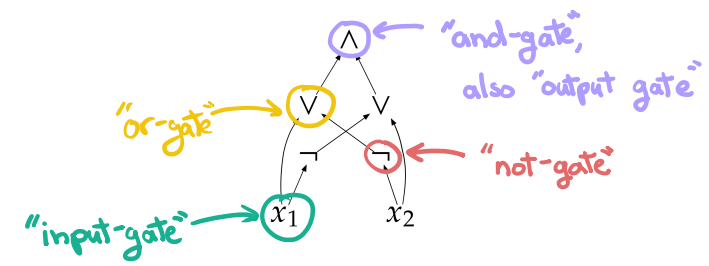
(definition)

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

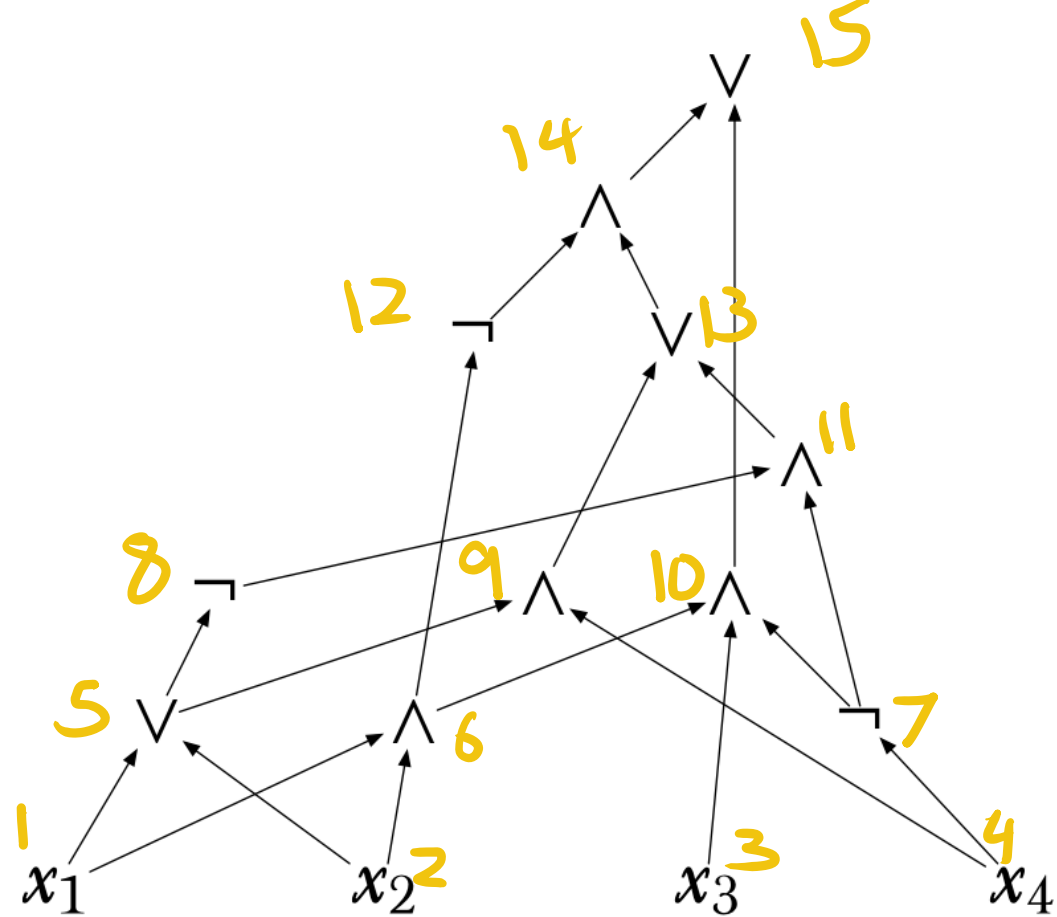
Topological order

think of arcs as " $\leq$ "

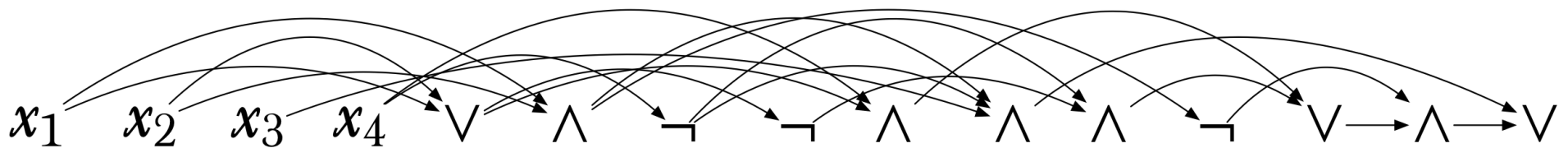
(directed edge)



this defines a
partial order on V



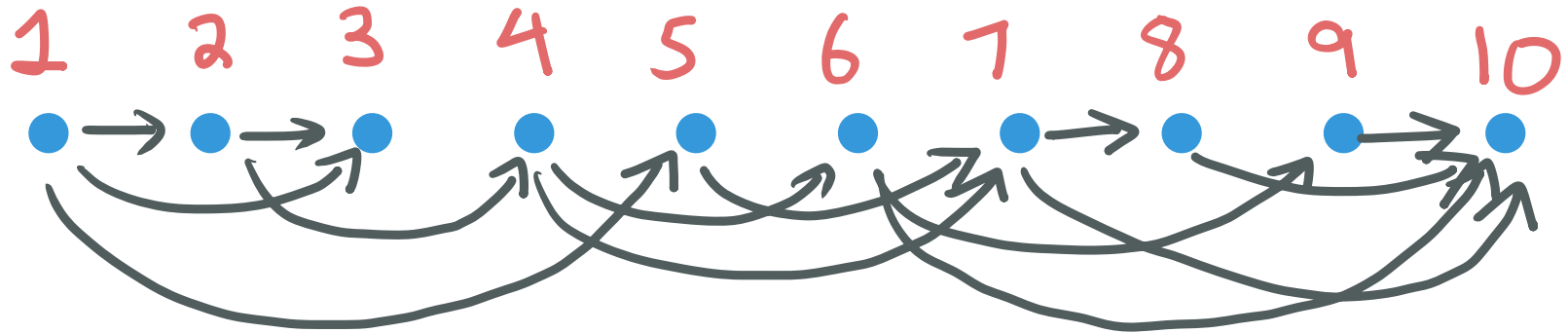
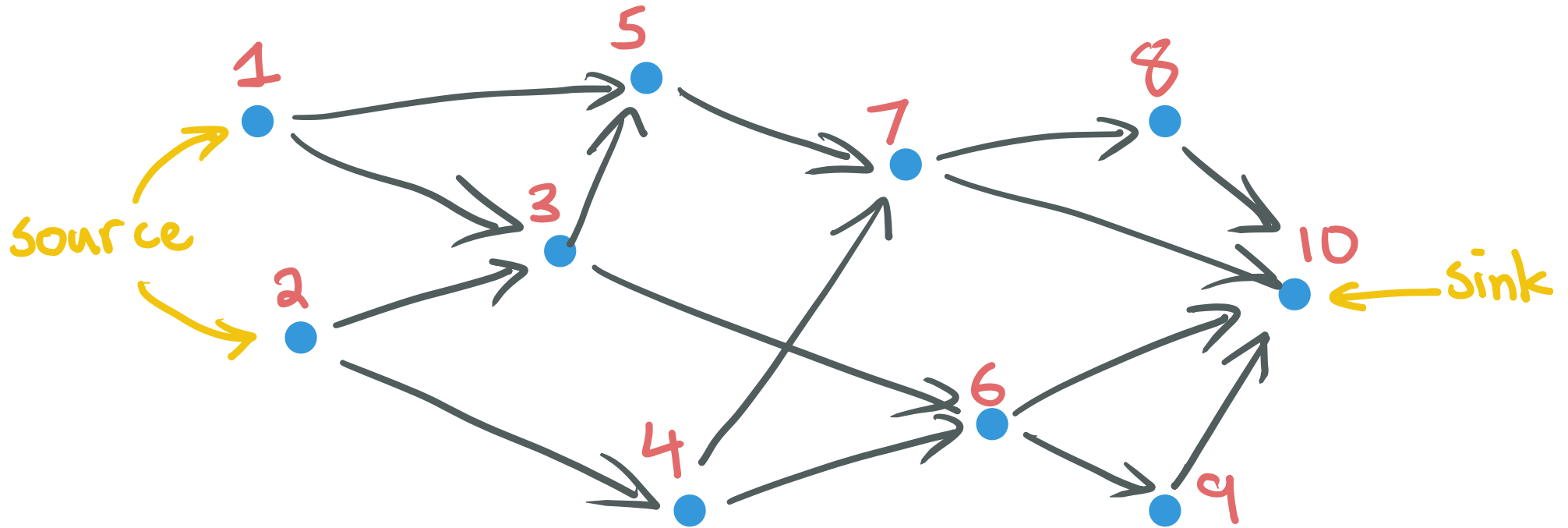
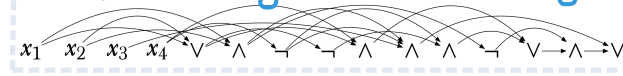
Topological ordering (of V):
any ordering consistent w/ edges



to evaluate, process gates from left to right!

How to topologically sort:
repeatedly remove sources

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

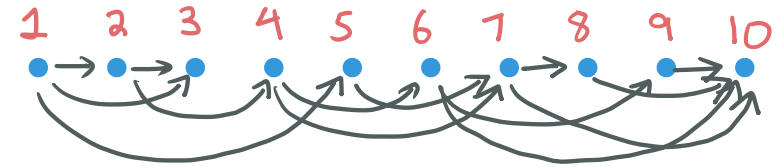
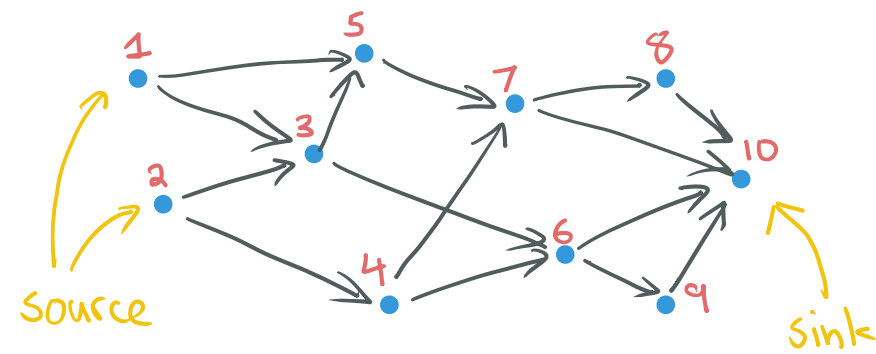


source-peeling($G = (V, E)$)

/ We assume G is a DAG.*

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

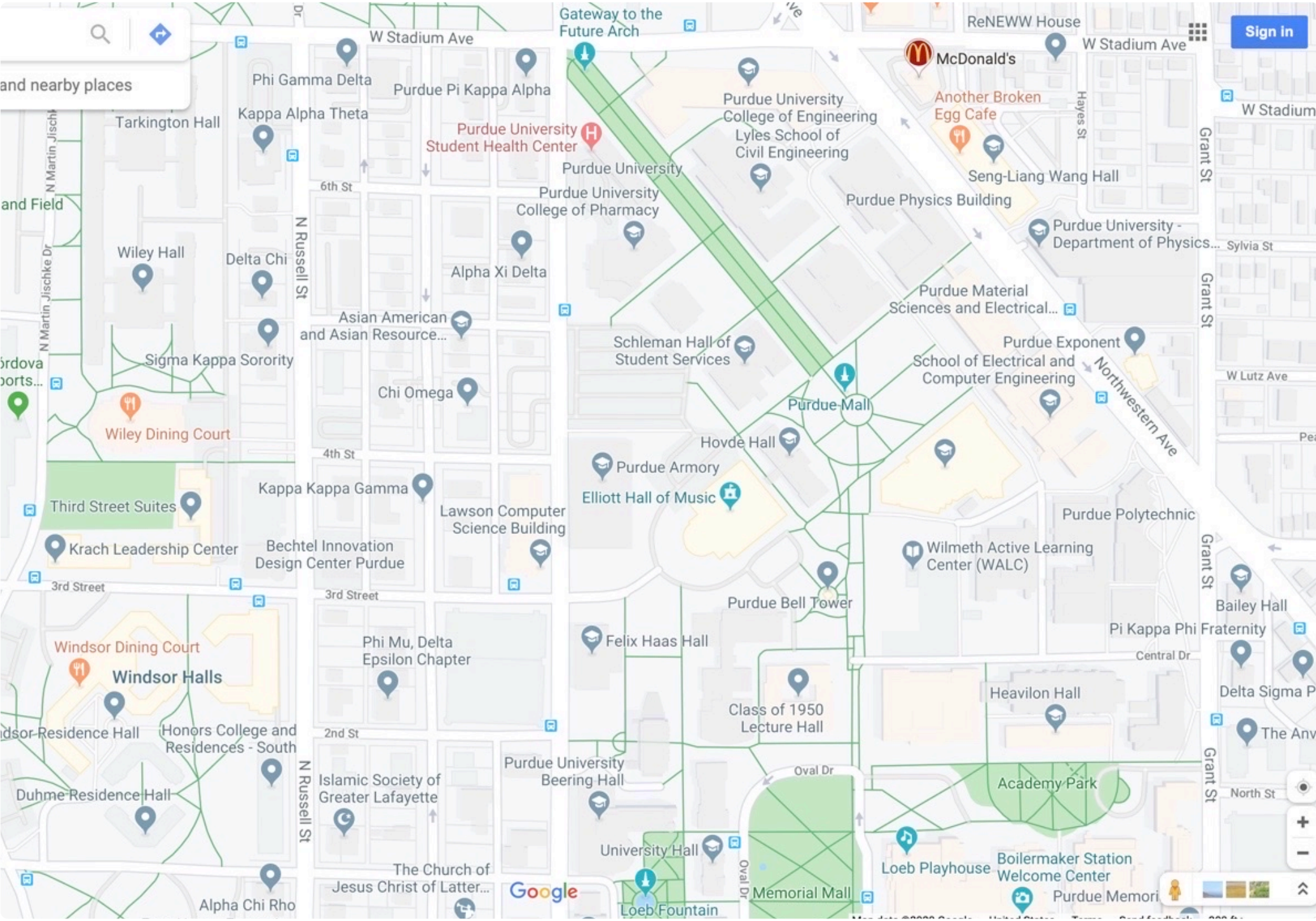


Theorem 6.1. Given a DAG G with m edges and n vertices, one can compute a topological ordering of V in $O(m + n)$ time.

I. DAG's and topological orderings

II. Searching graphs

III. Sorting graphs



and nearby places

Sign in

Google

CAT MAZE

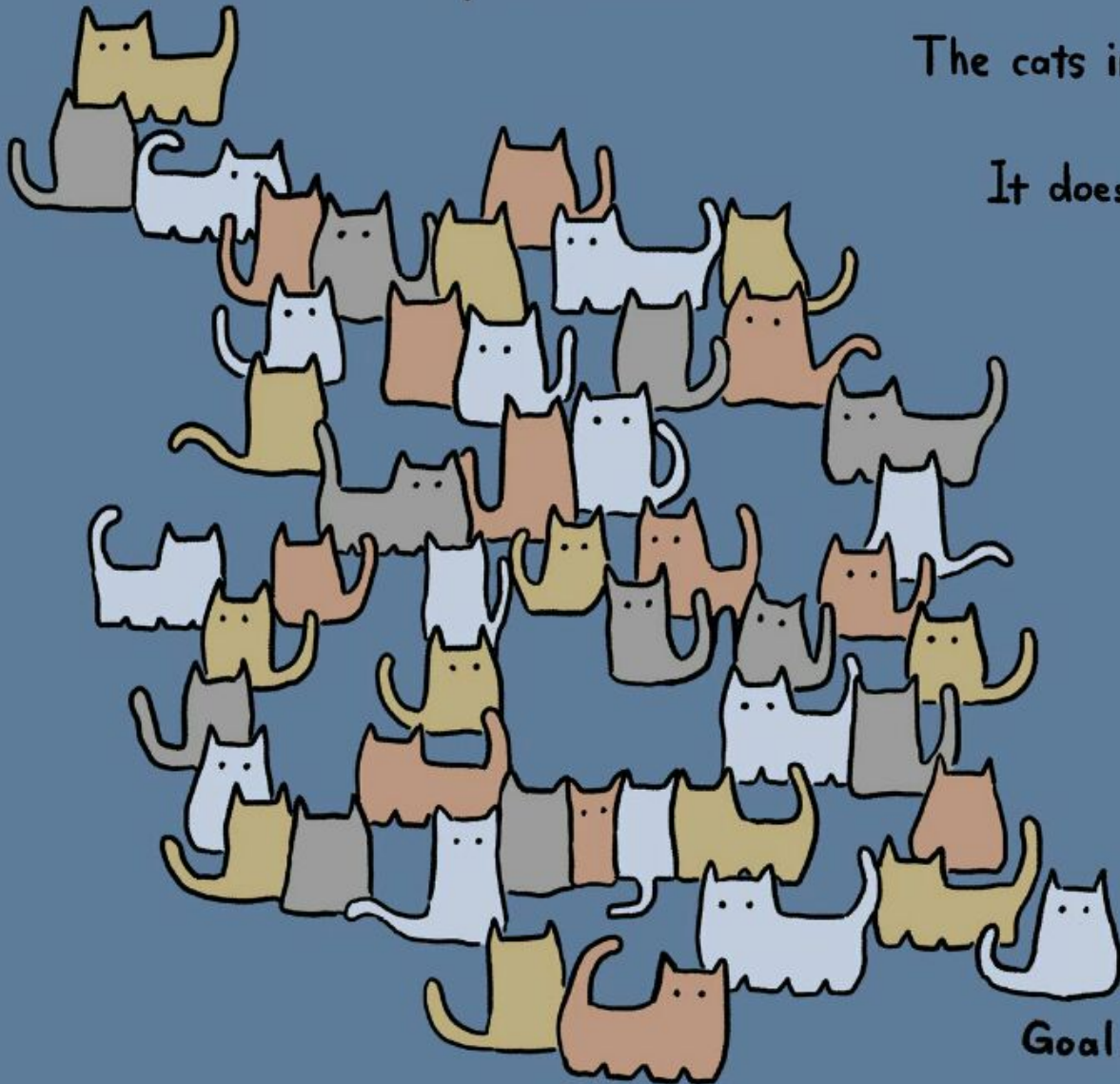
by Tatsunami

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

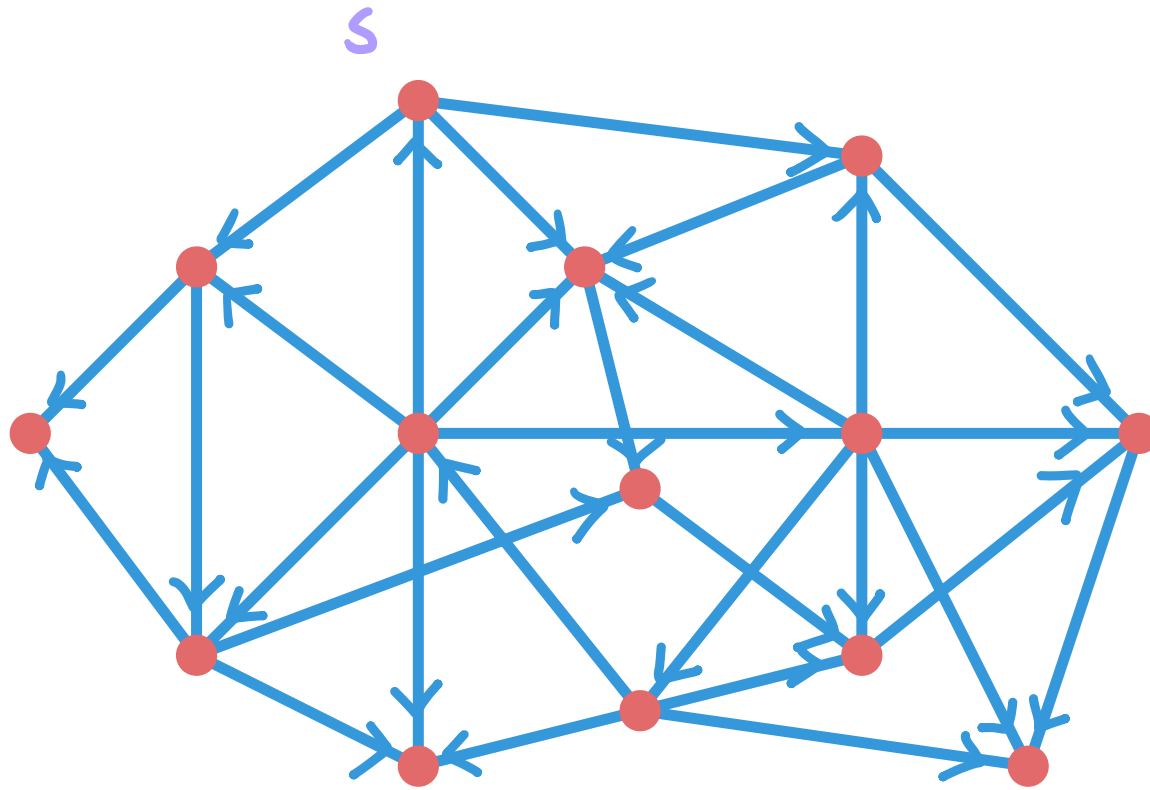
The cats in your path must be connected or overlapping.

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

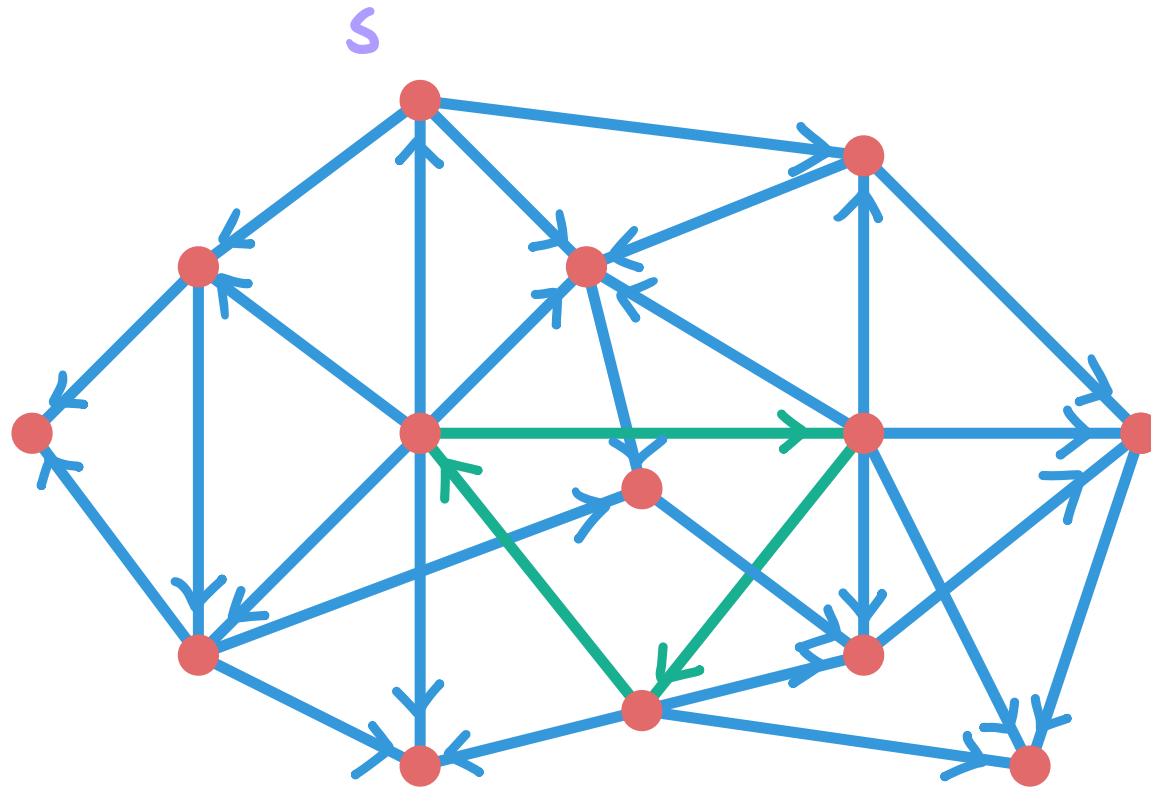
Start



Goal

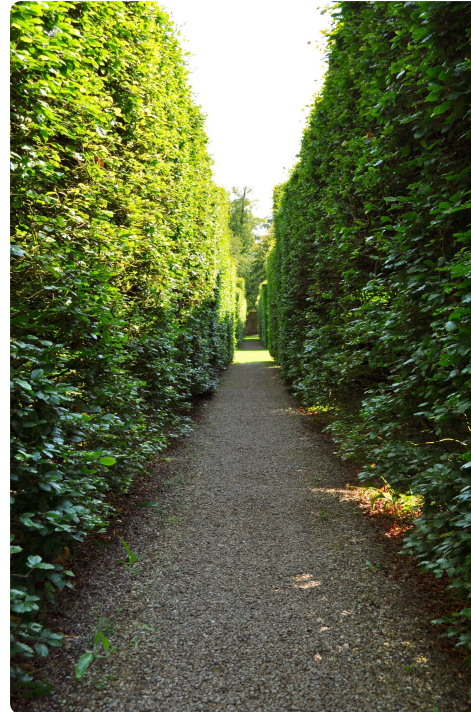
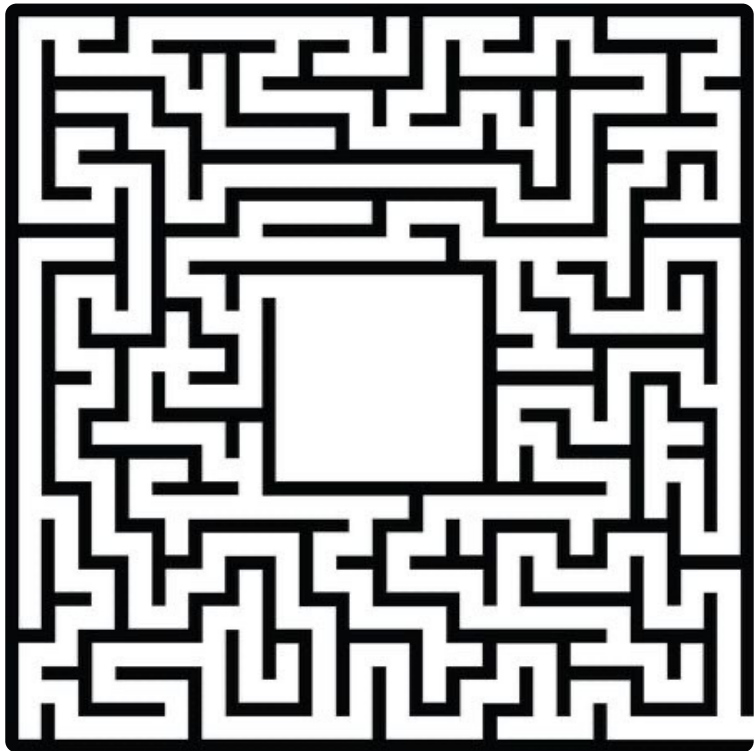
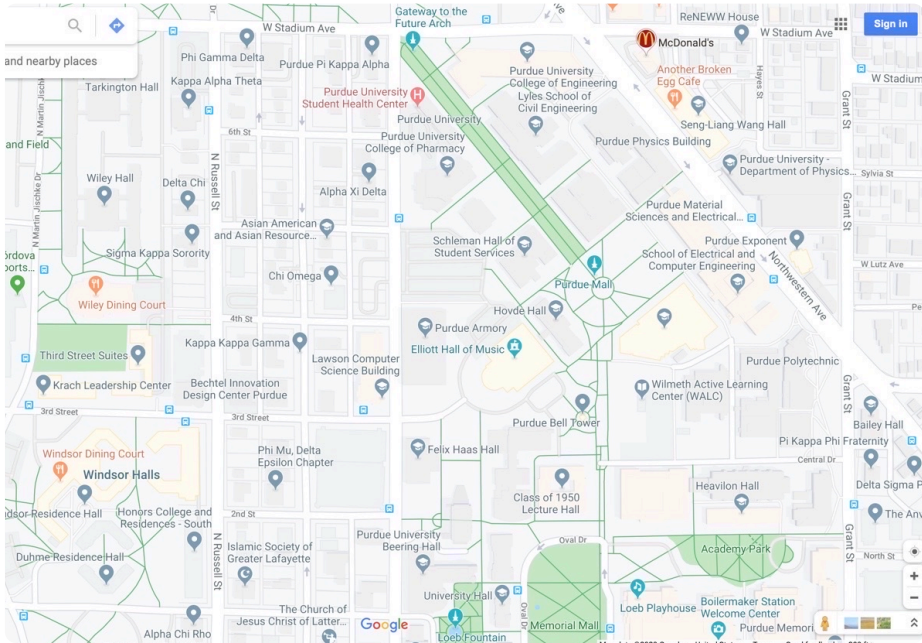


can s reach all the vertices?



can s reach all the vertices? (no)

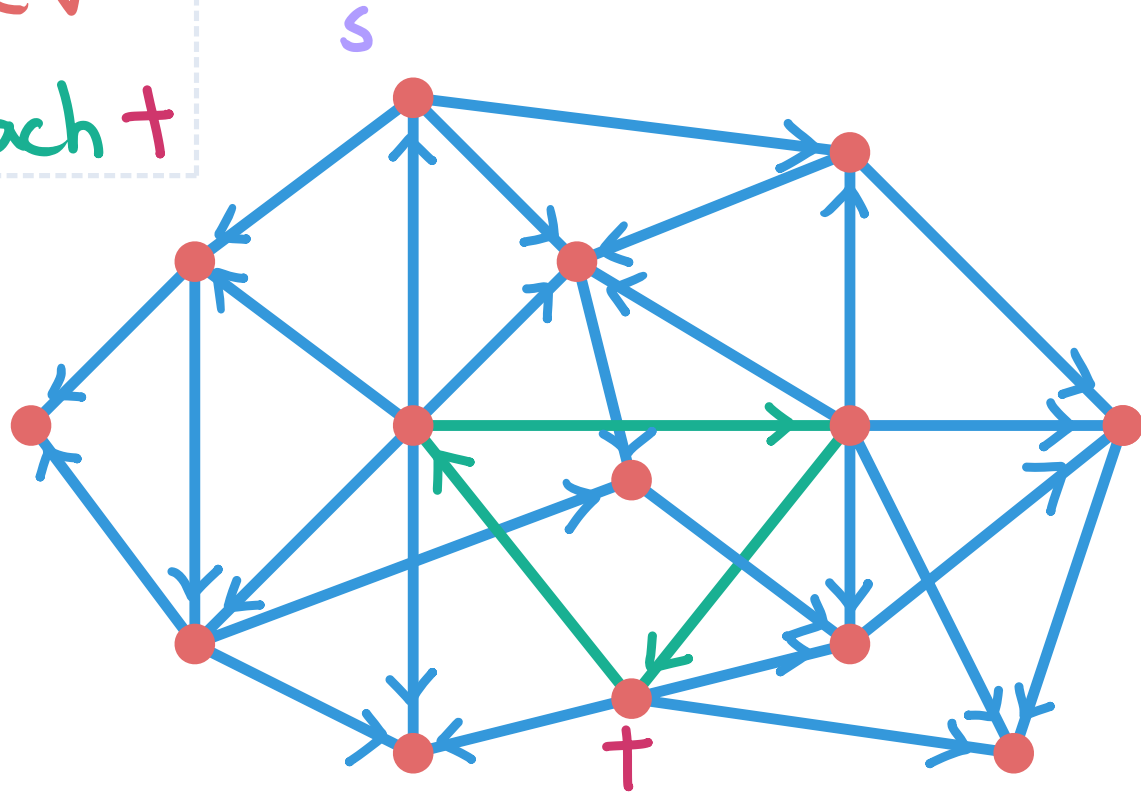
Goal: design an algorithm to figure this out automatically



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

Goal: decide if s can reach t

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



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

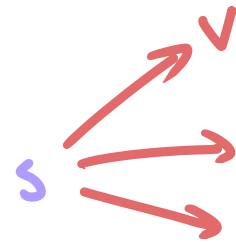
1. if $s=t$ then return TRUE

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

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

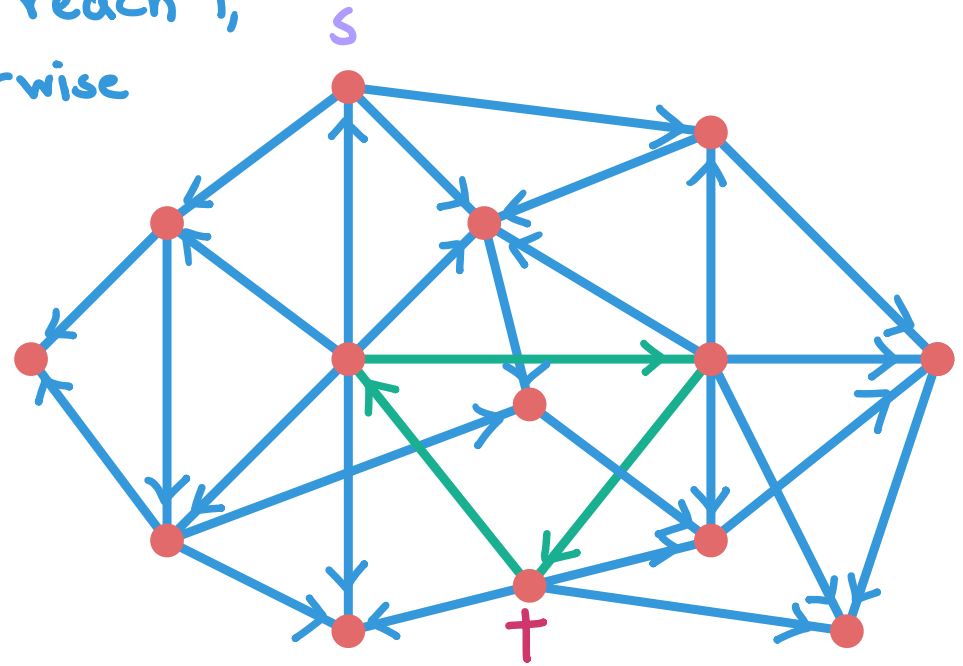
1. if $s=t$ then return TRUE

2. for each edge (s,v)

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

then return TRUE

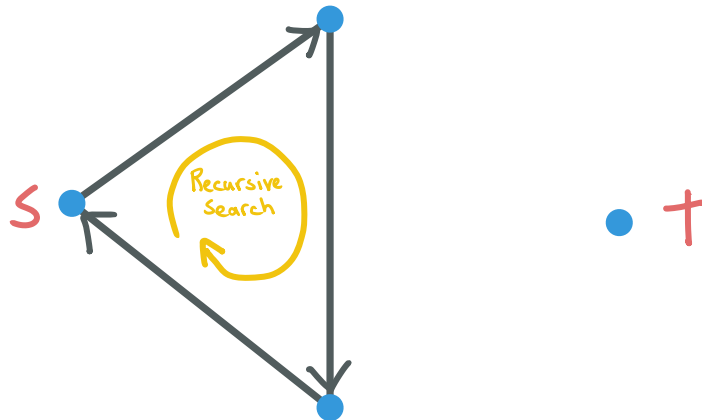
3. return FALSE



Argument: if s can reach t , and $s \neq v$, 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

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

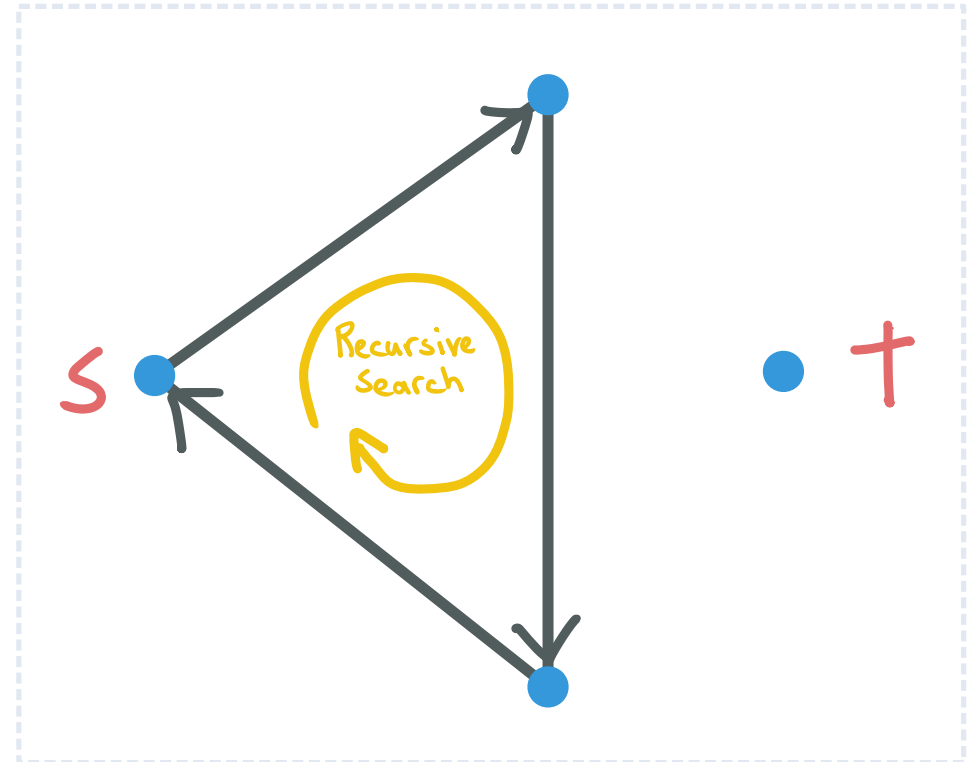
1. if $s = t$ then return TRUE

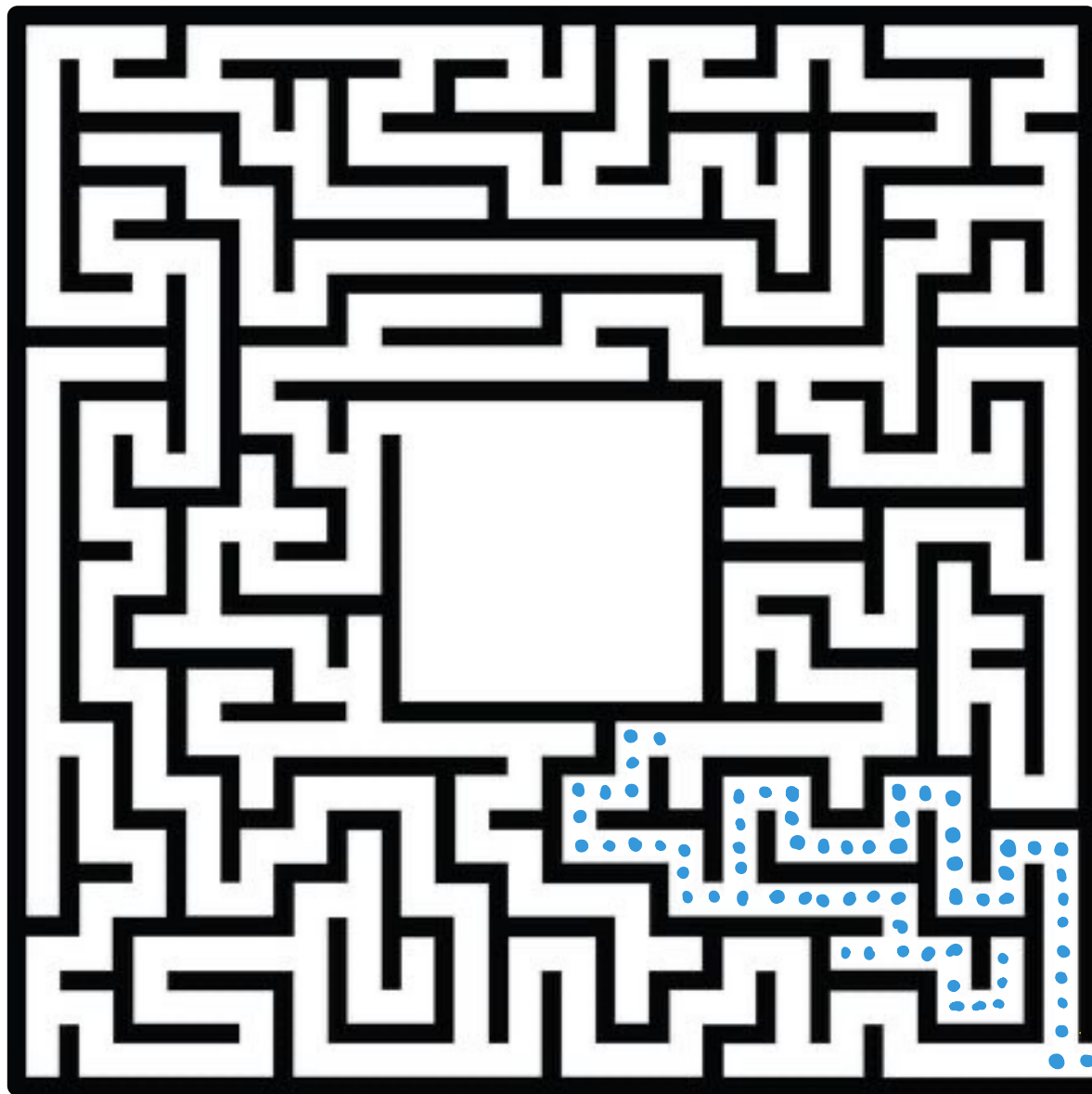
2. for each edge (s, v)

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

then return TRUE

3. return FALSE





Breadcrumbs

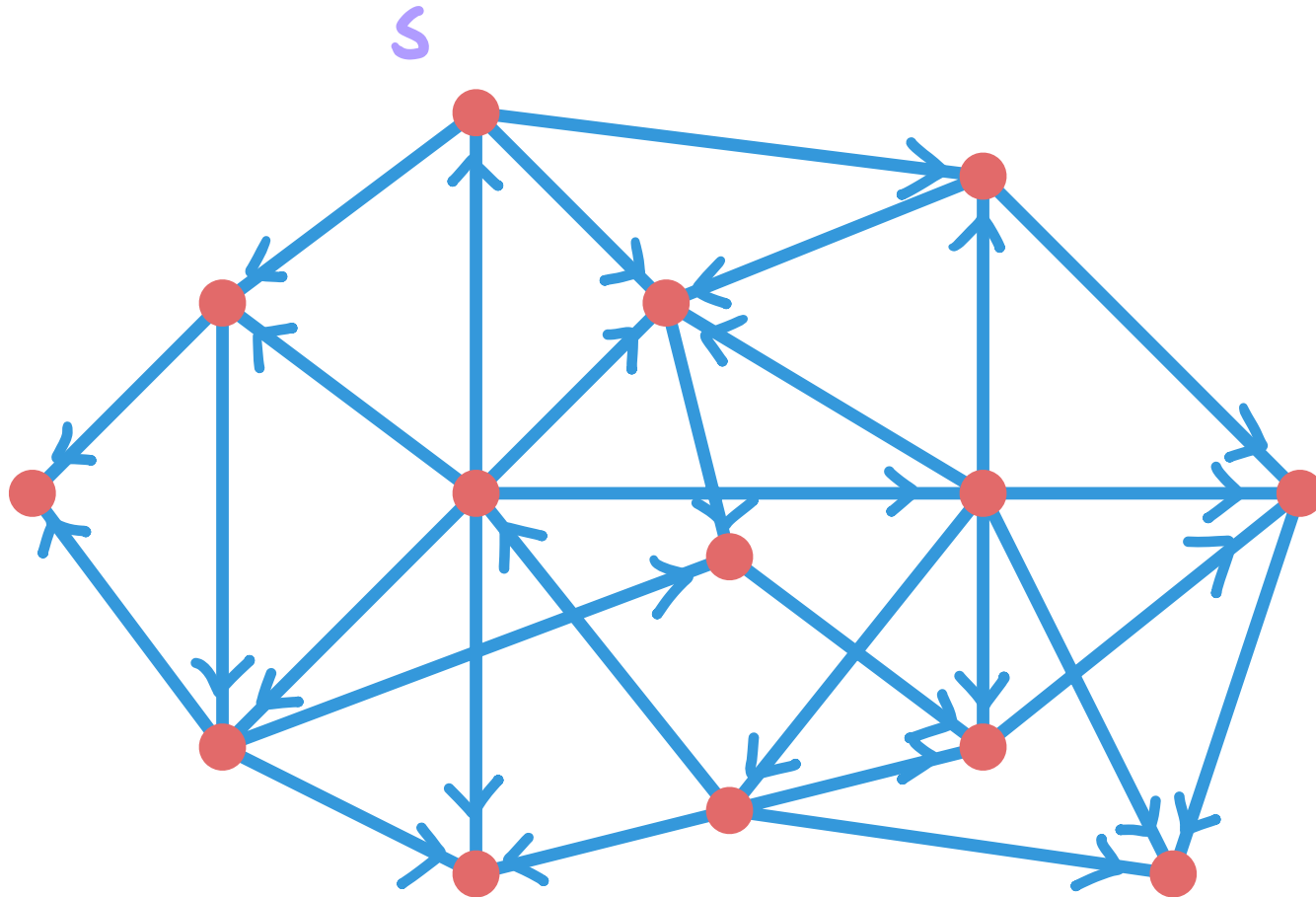
DFS(v)

1. if v is unmarked

A. mark v

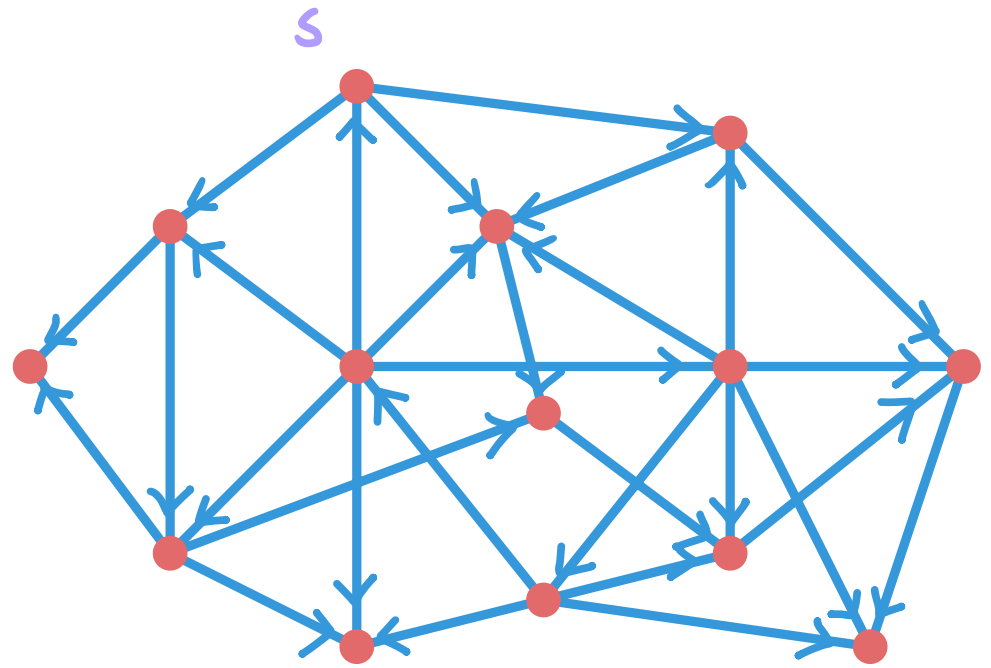
B. for each edge (v, w) from v

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.

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

we mark each vertex once

$\Rightarrow$ traverse each edge once

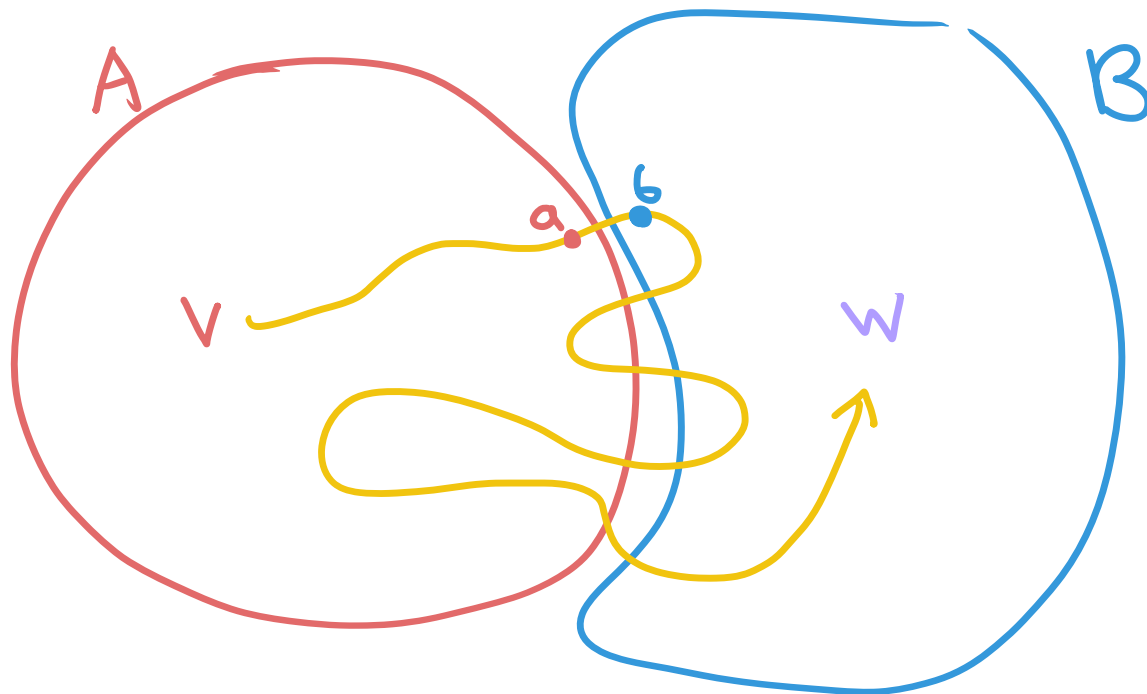
② $\text{DFS}(v)$ visits all vertices reachable from v

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

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

let w be reachable from v

suppose by contradiction that $w \in B$



$\text{DFS}(v)$

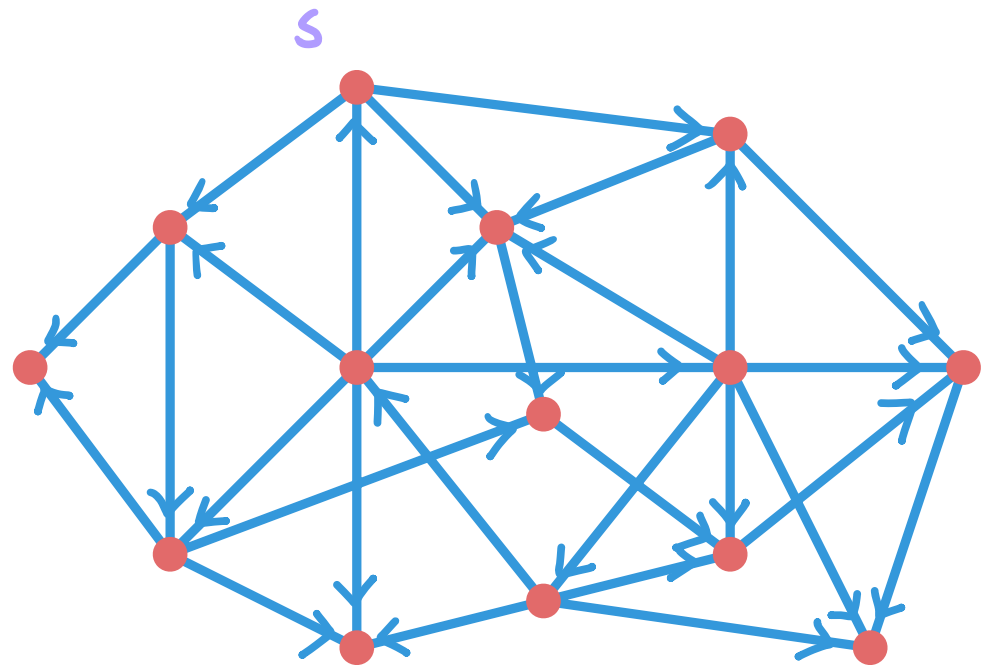
1. if v is unmarked
 - A. mark v
 - B. for each edge (v, w) from v
 1. $\text{DFS}(w)$

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

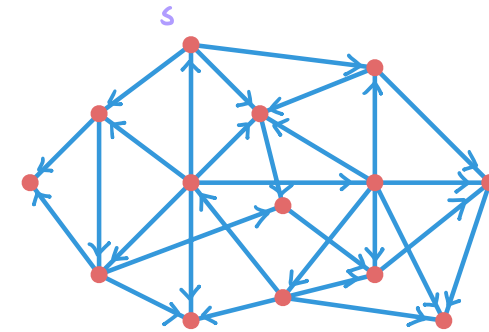
DFS(v)

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

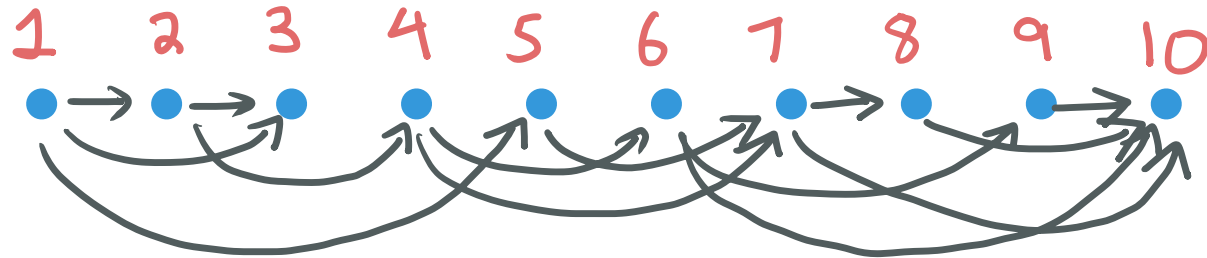
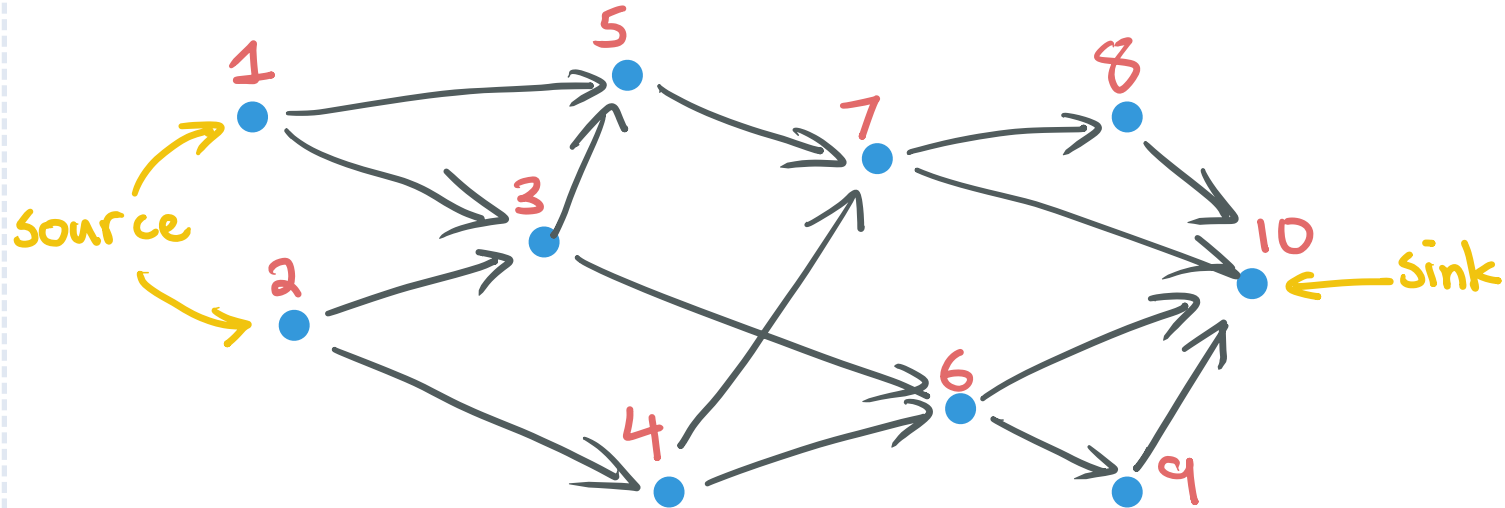
II. Searching graphs

III. Sorting graphs

Recall:

How to topologically sort:
repeatedly remove sources

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

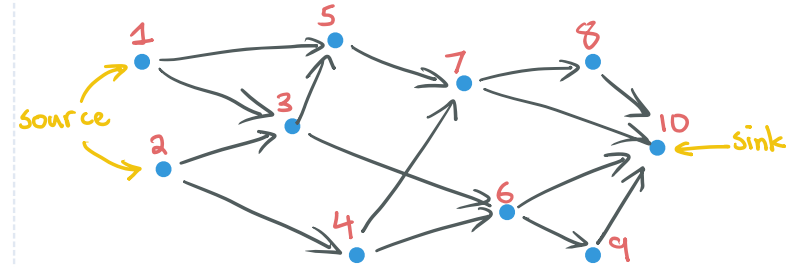


DFS(v)

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 $\text{DFS}(v)$

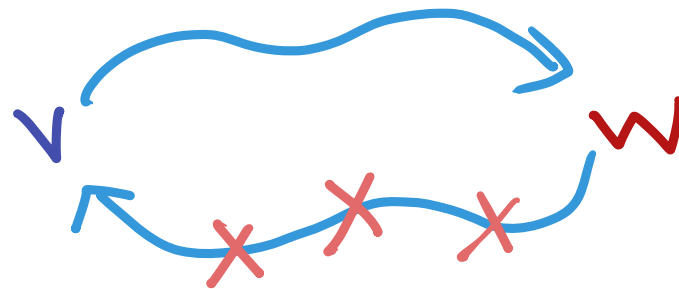
$\text{DFS}(v)$

1. if v is unmarked

A. mark v

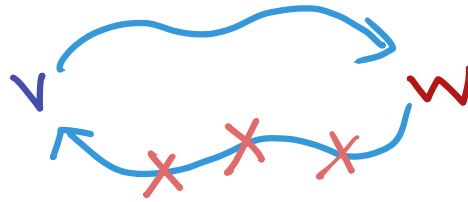
B. for each edge (v, w) from v

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 w .

Case 2: DFS marks w first

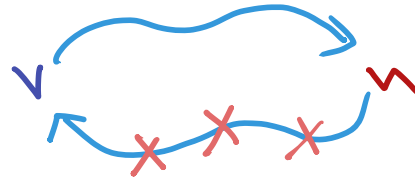
then DFS(w) only searches vertices

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

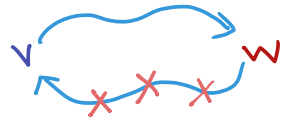
post-DFS-driver($G = (V, E)$)

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

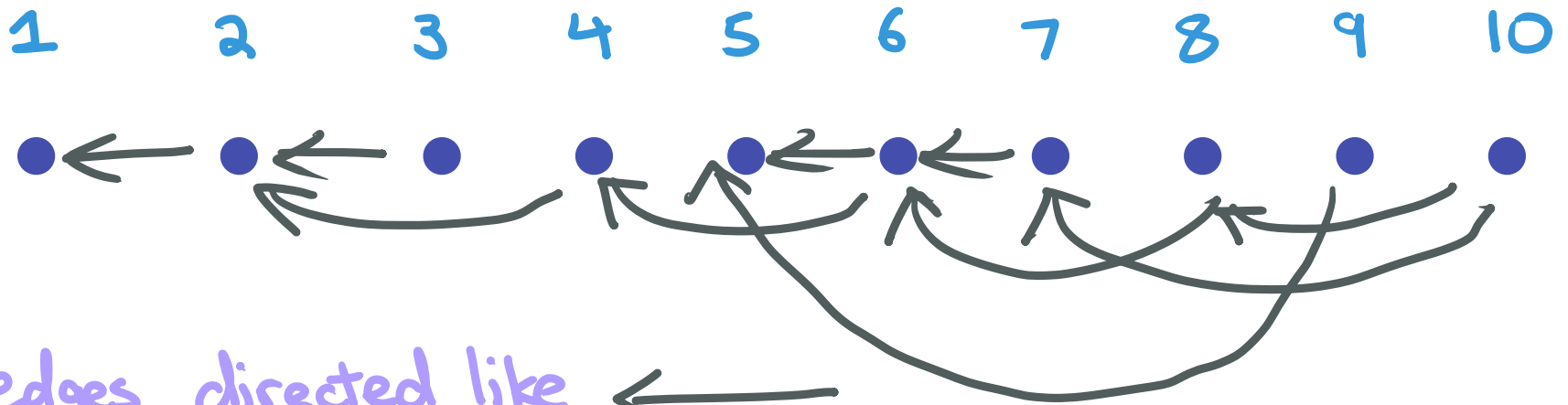
/ The following line is the only difference*

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

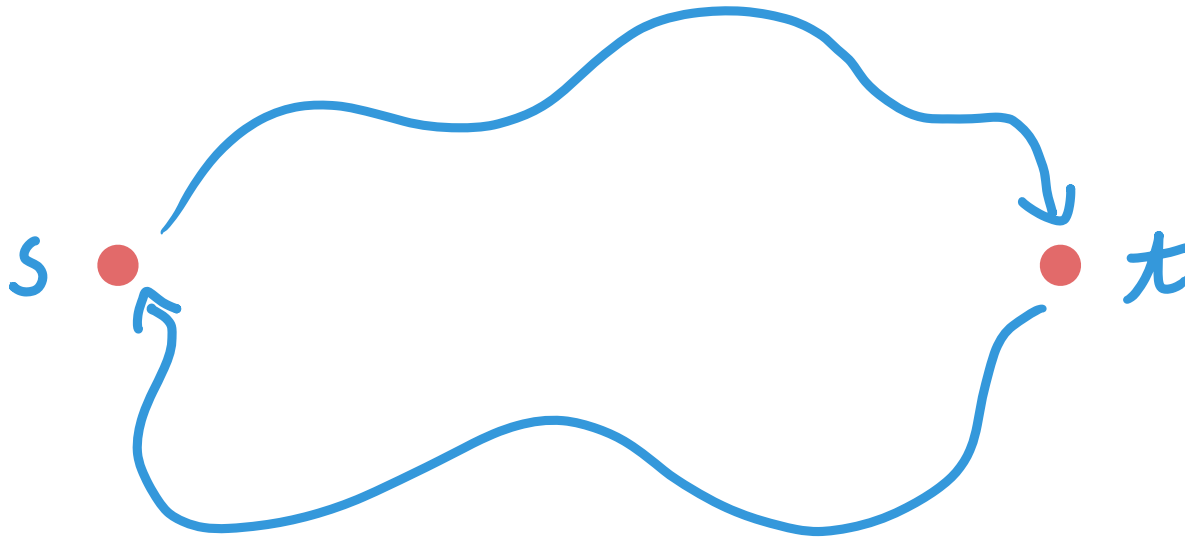
Recall:

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

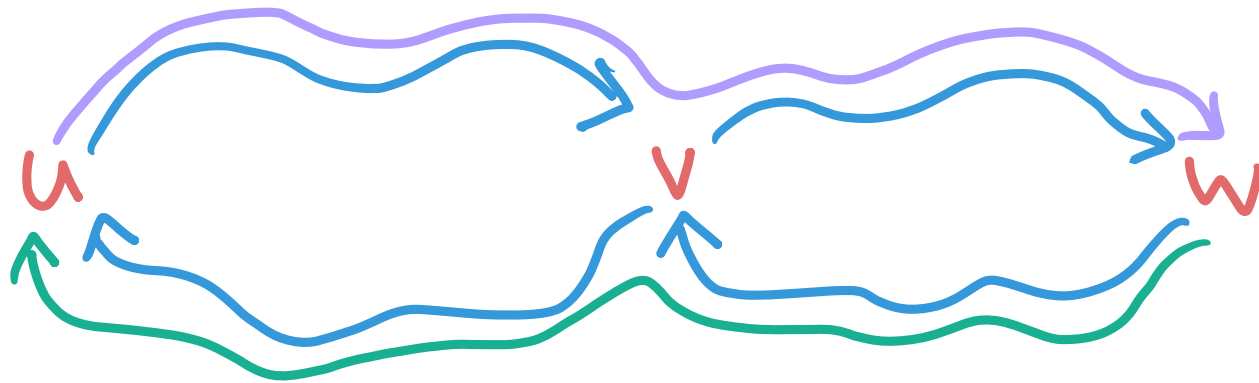


a graph is strongly connected if every pair
of vertices is strongly connected

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)

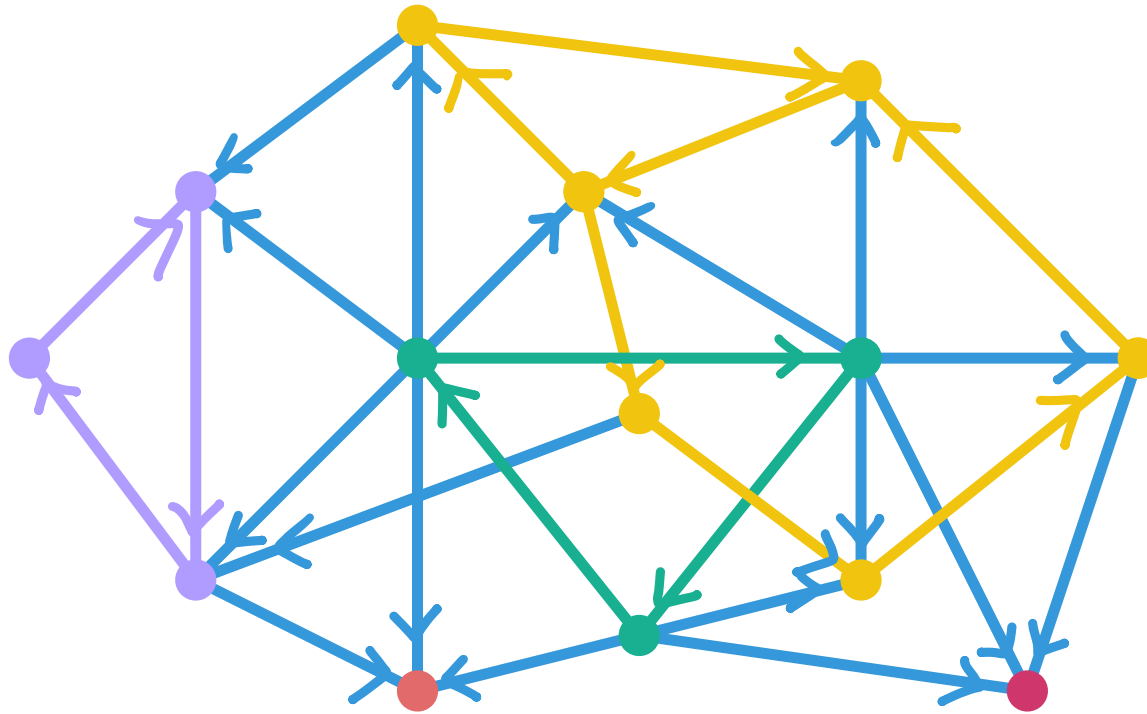
Proof



combine paths from $u \rightsquigarrow v$, $v \rightsquigarrow w$ to get path
from $u \rightsquigarrow w$

similar for other direction

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



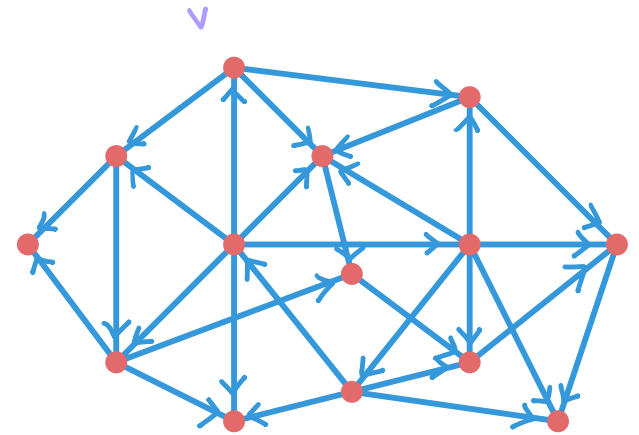
Goal: find the SCC's algorithmically
(strongly connected components)

First:

given v , find the SCC of v

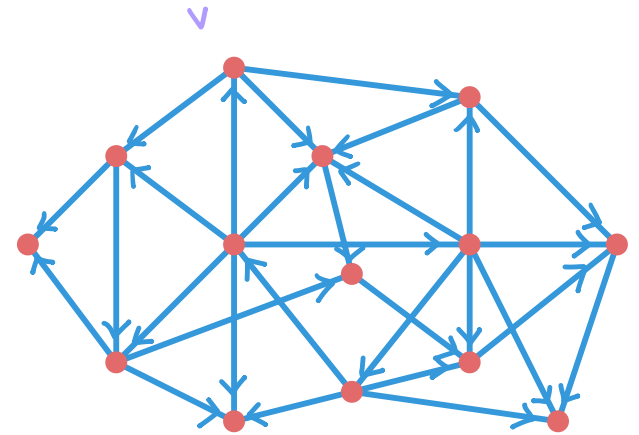
(strongly
connected
component)

ideas?



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



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

$SCC(v)$

① run DFS from v

$O(m+n)$

② run DFS from v in
the reversed graph

$O(m+n)$

③ return set of vertices
visited in both DFS

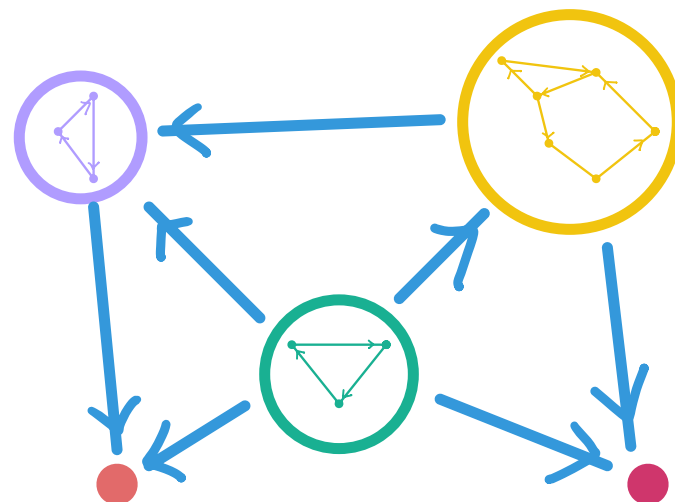
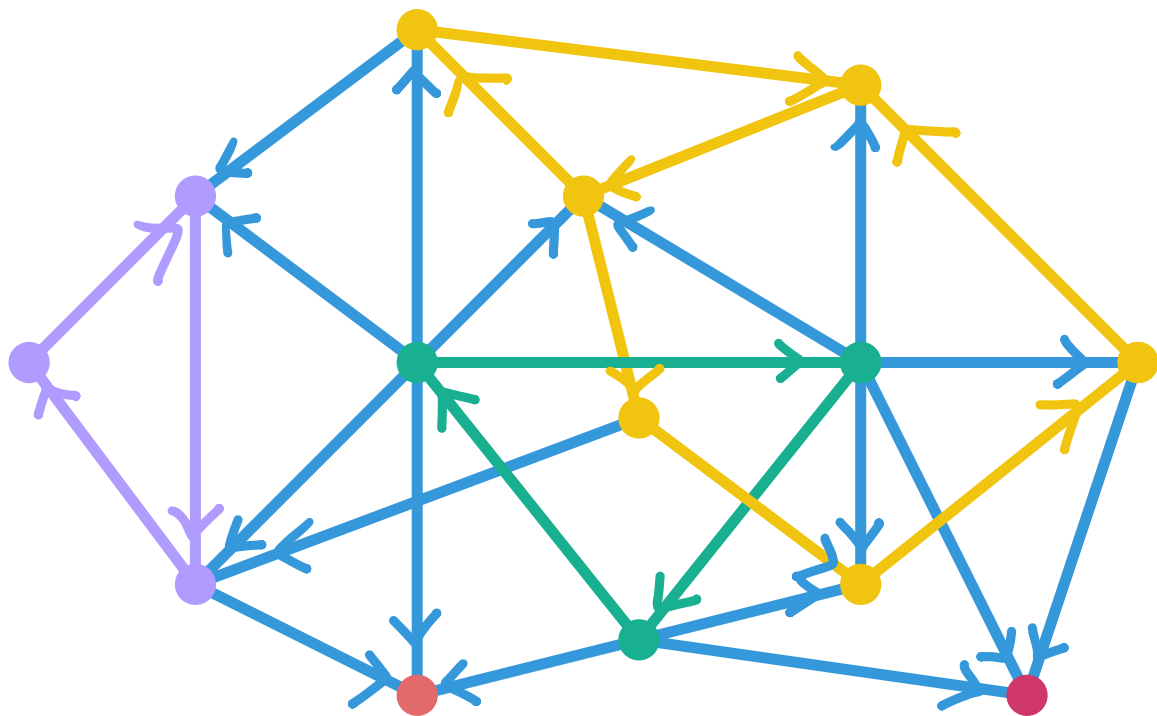
$O(m+n)$

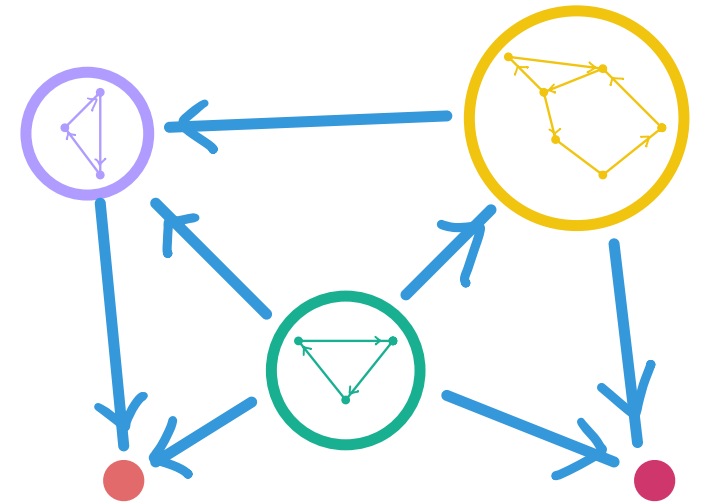
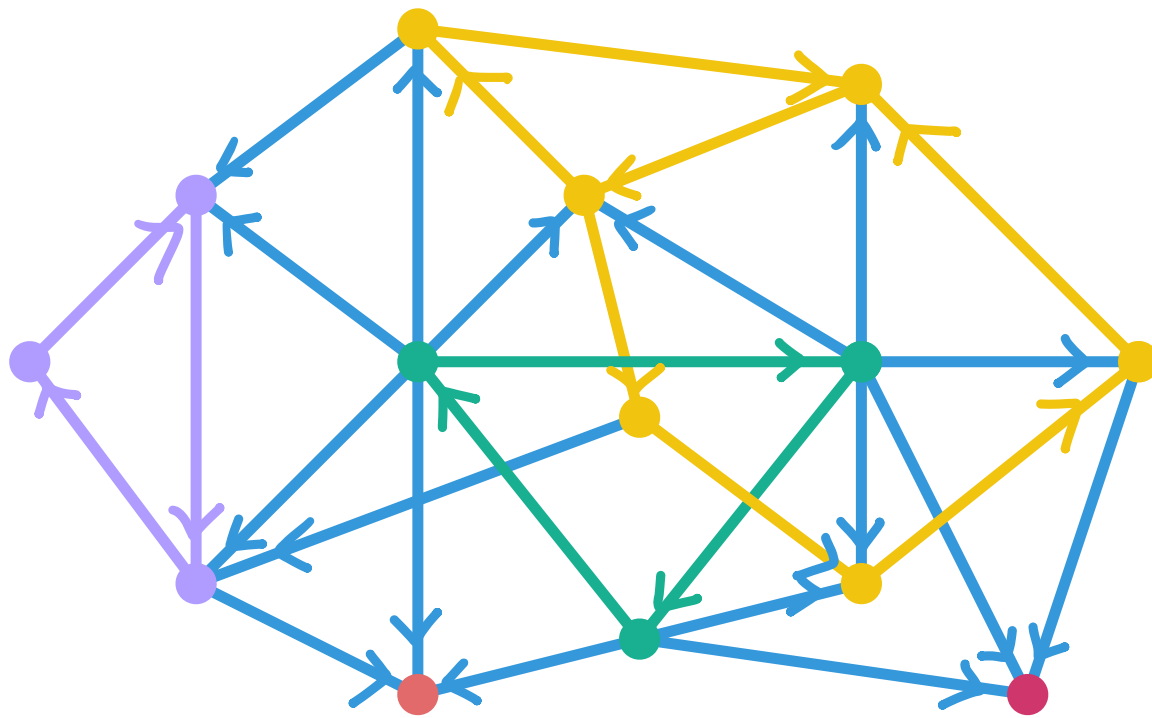
to get all the SCC's:

run from every vertex

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

better?





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

Our current approach

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

1. until $V=\emptyset$

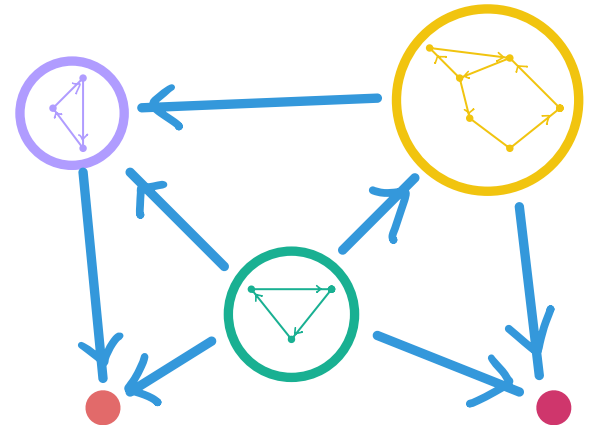
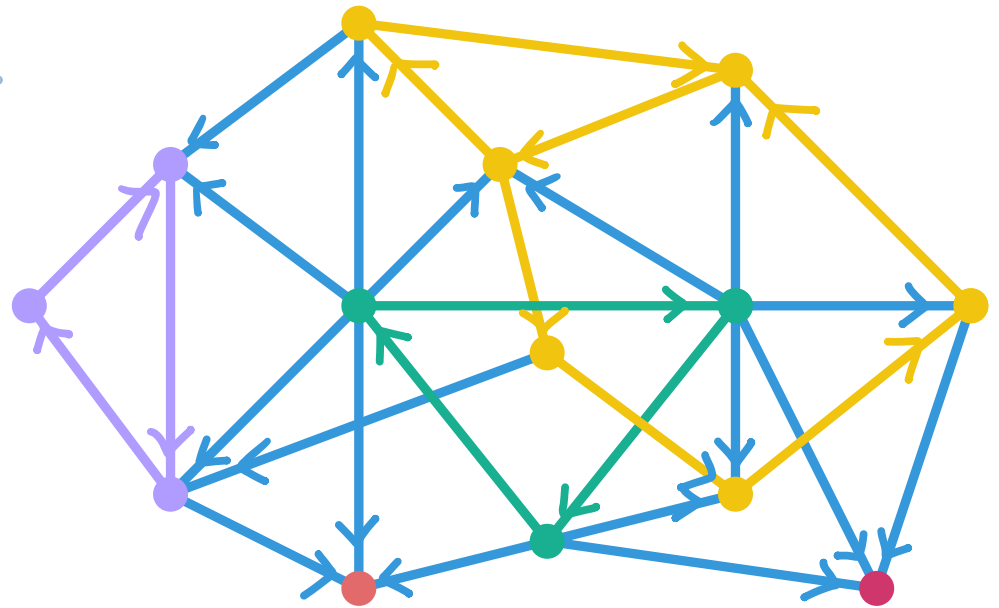
a. let v be any vertex in V

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 $O(mn)$)



sink-first

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

we don't know
how to do this
yet

until $V=\emptyset$

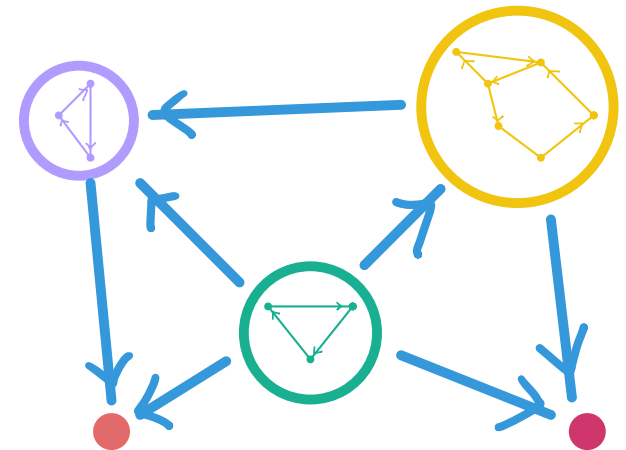
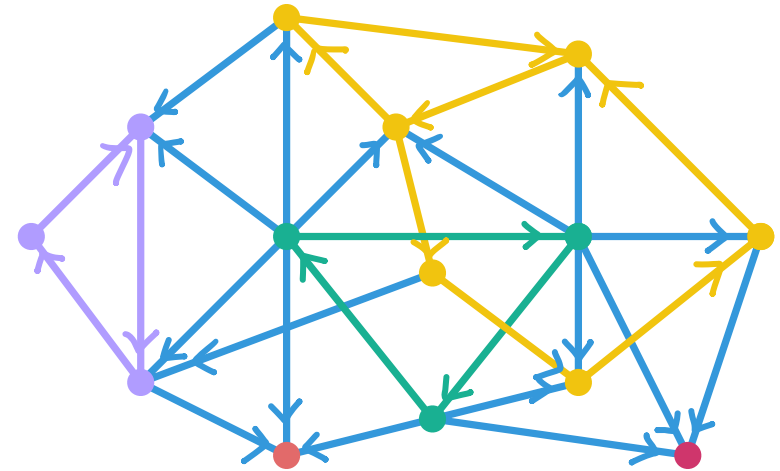
let v be any vertex ~~in V~~
in a sink component

DFS

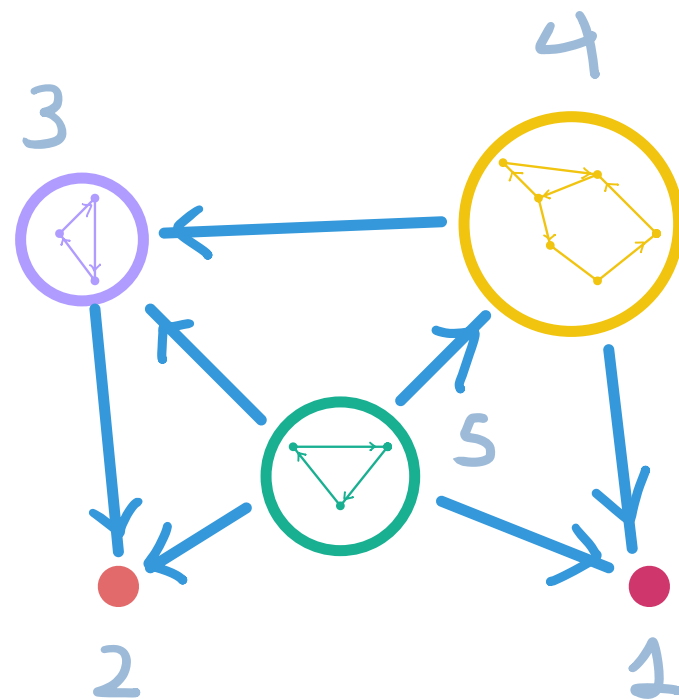
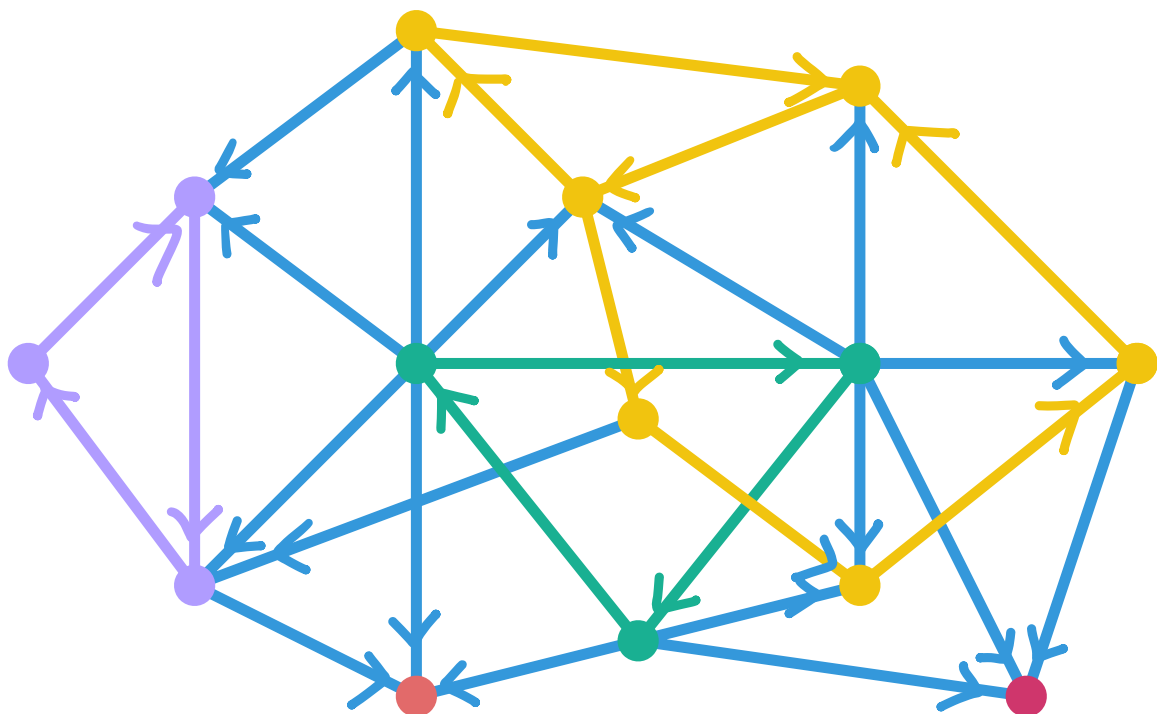
$C \leftarrow \text{SCC}(v)$ // DFS cannot leave sink
label each vertex in C w/ component

(some label for) C

remove C and all incident
edges from G

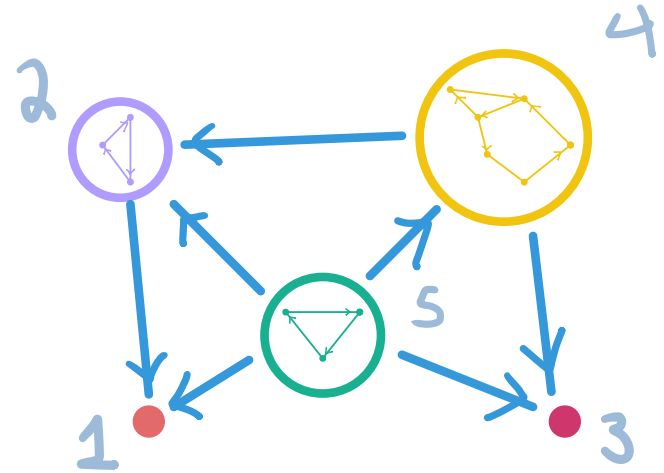
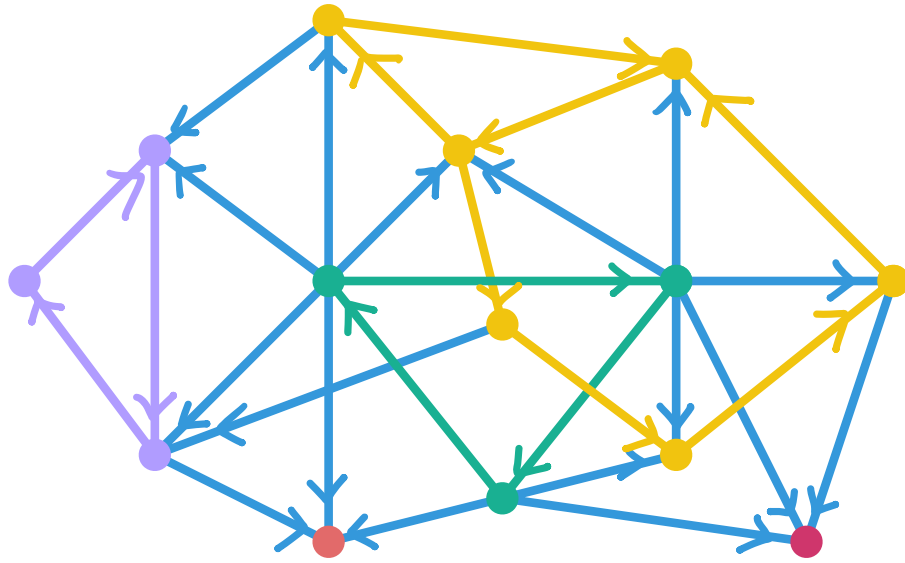


we only visit each vertex once!



To wit:

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



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

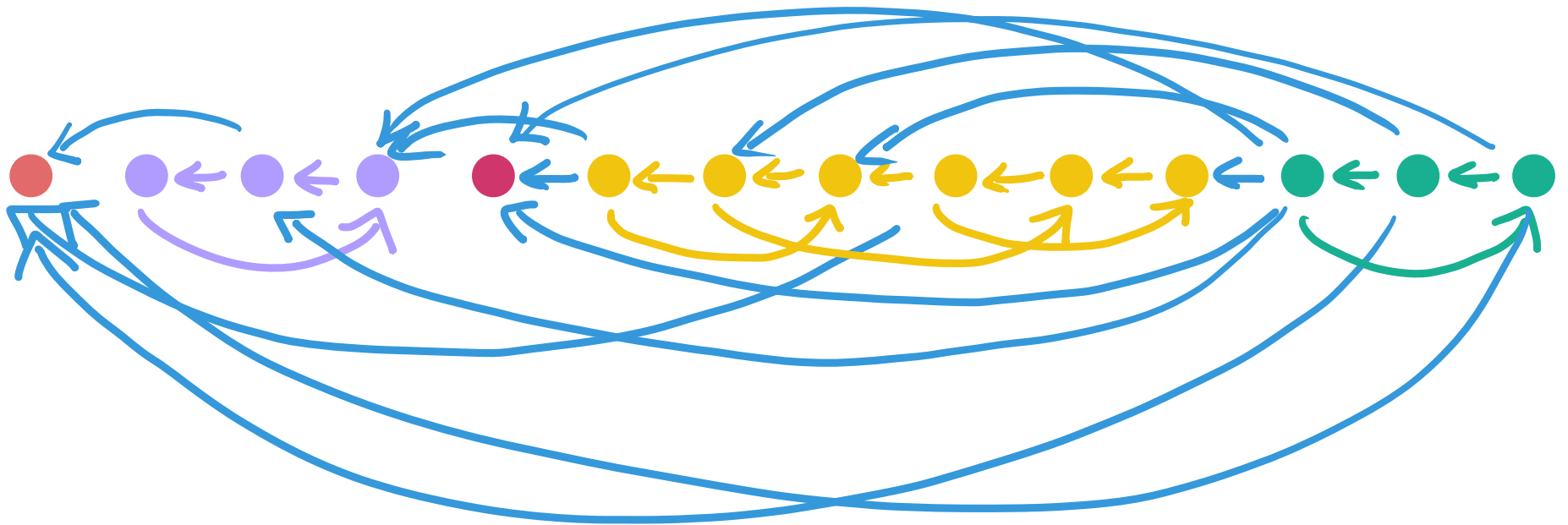
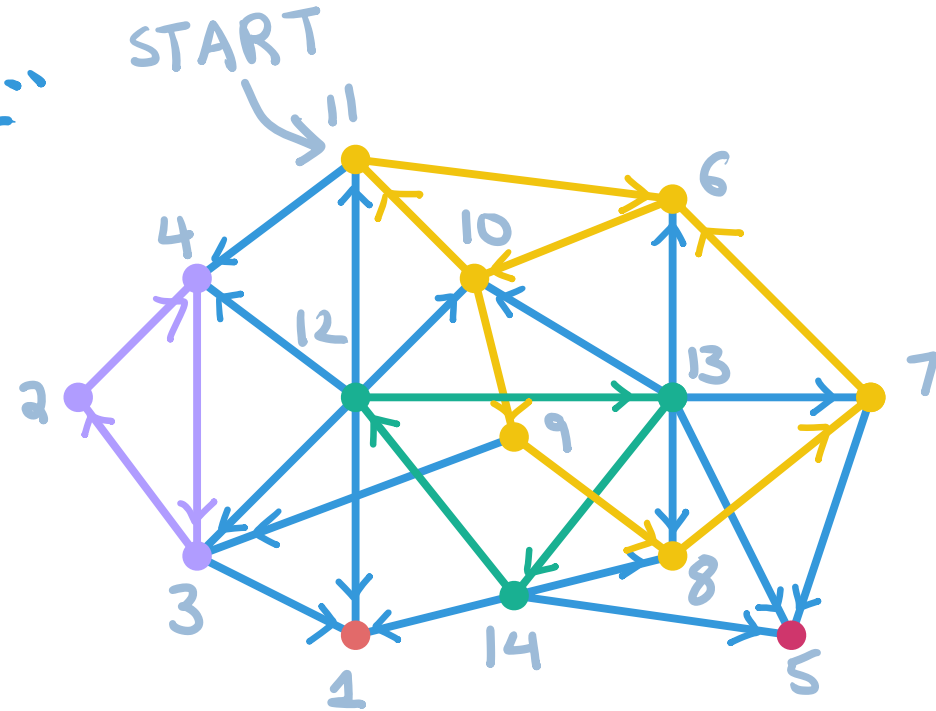
(b) $O(m+n)$ time total:

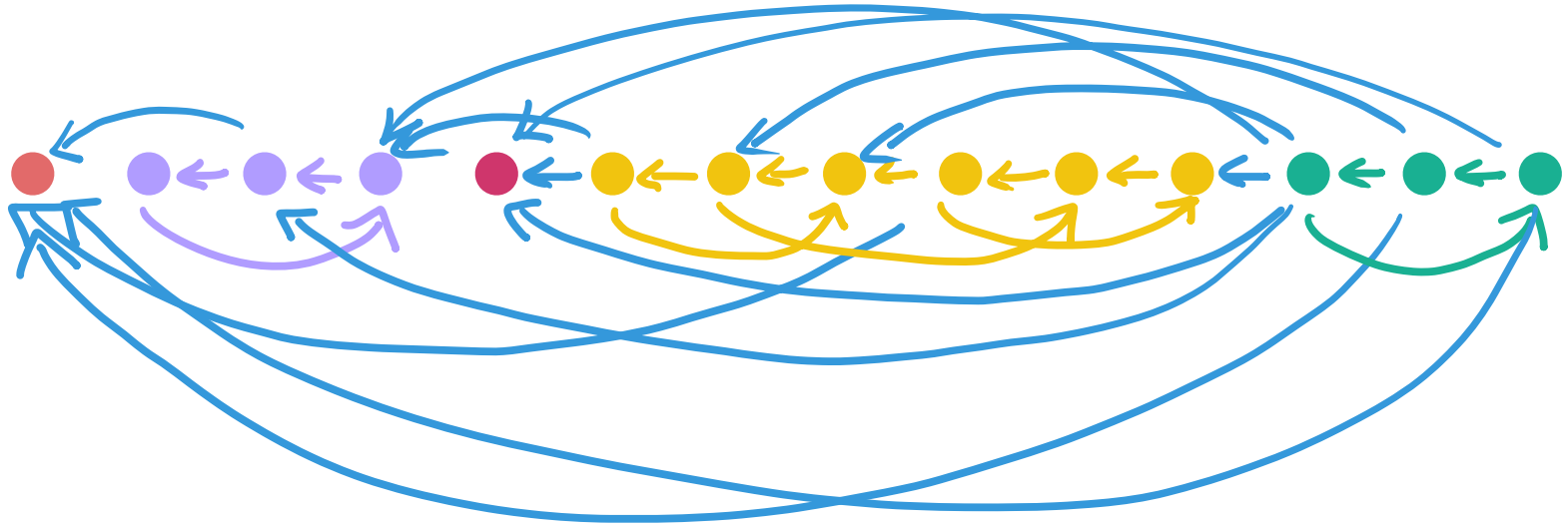
each $\text{DFS}(v)$ does not leave v 's ^(strongly connected component) SCC

how to find "sink-component" ordering?

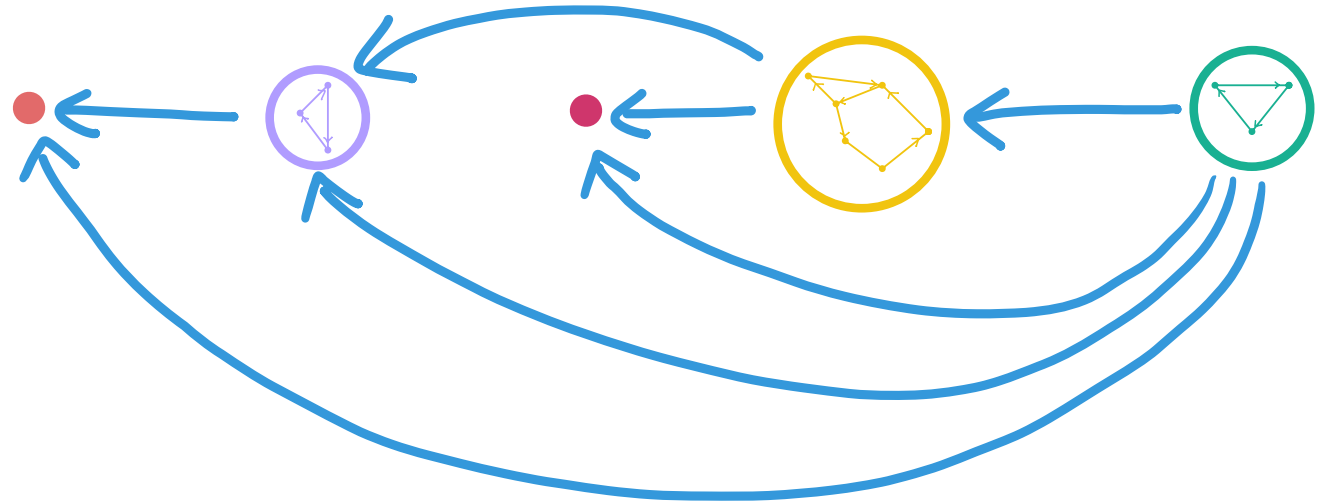
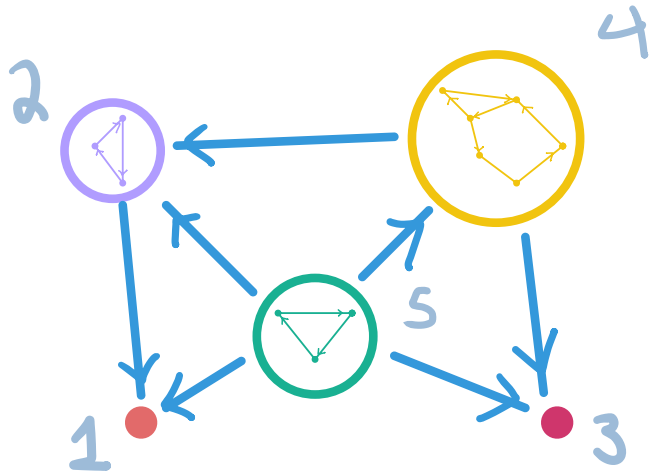
post-order DFS order

lists vertices in "sink-first order"





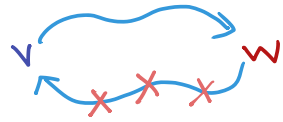
all cross-component edges [in BLUE] are in the same direction



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 .

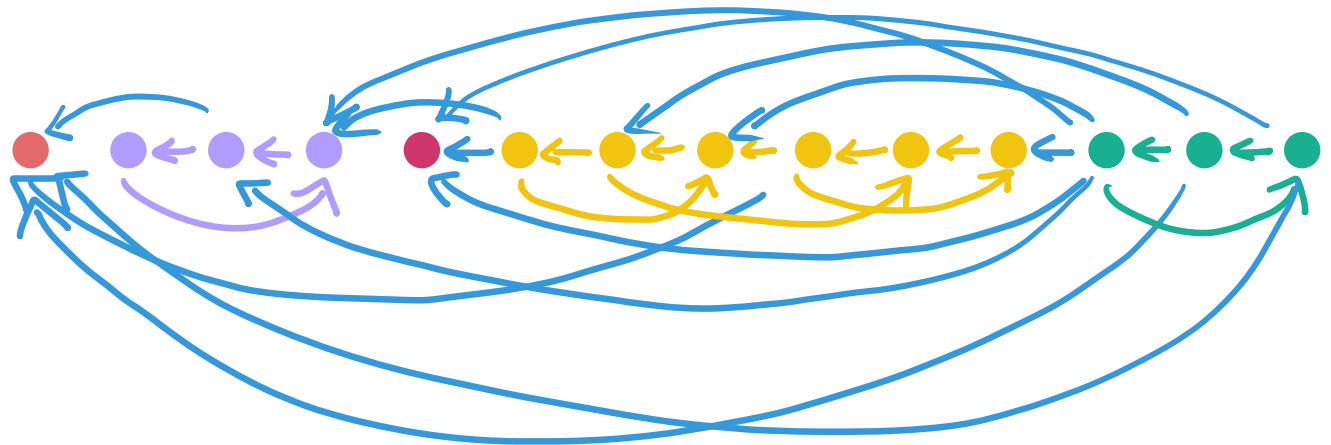
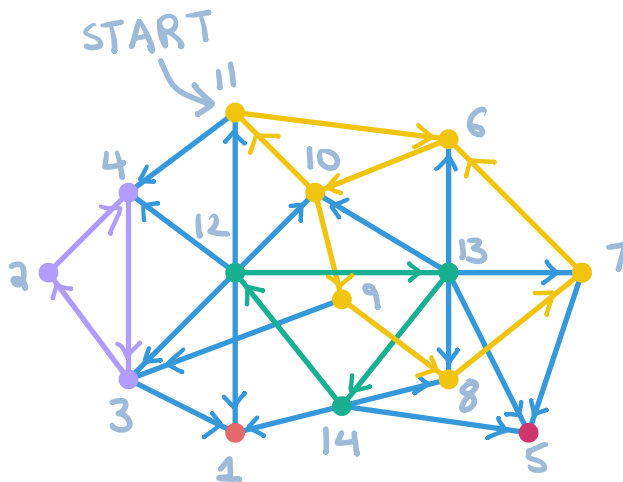
B. For each edge (v, w) from v :

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 $V=\emptyset$

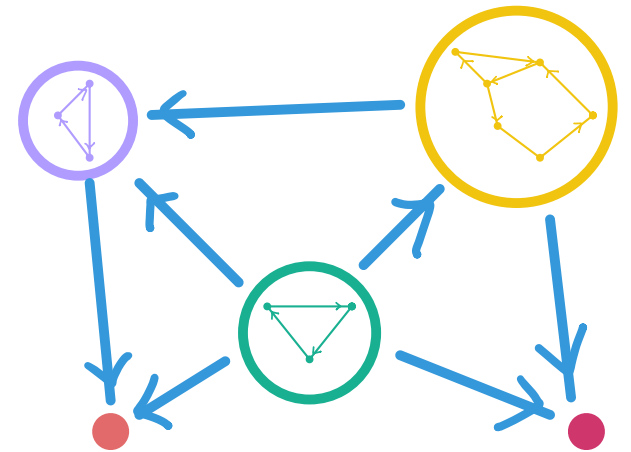
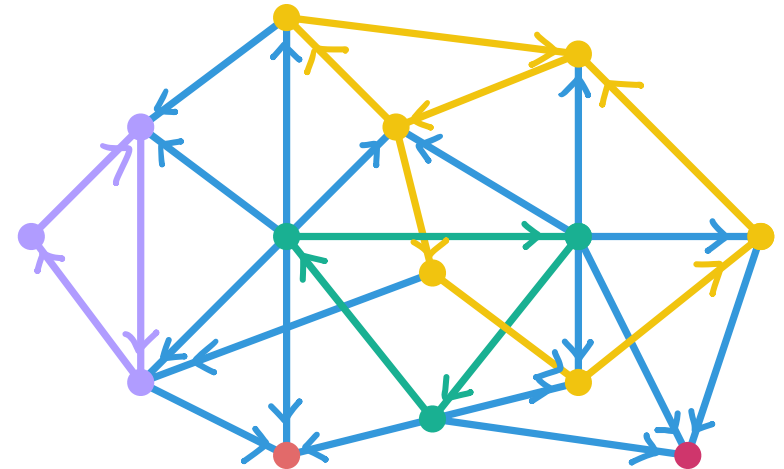
let v be a vertex in a sink component

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

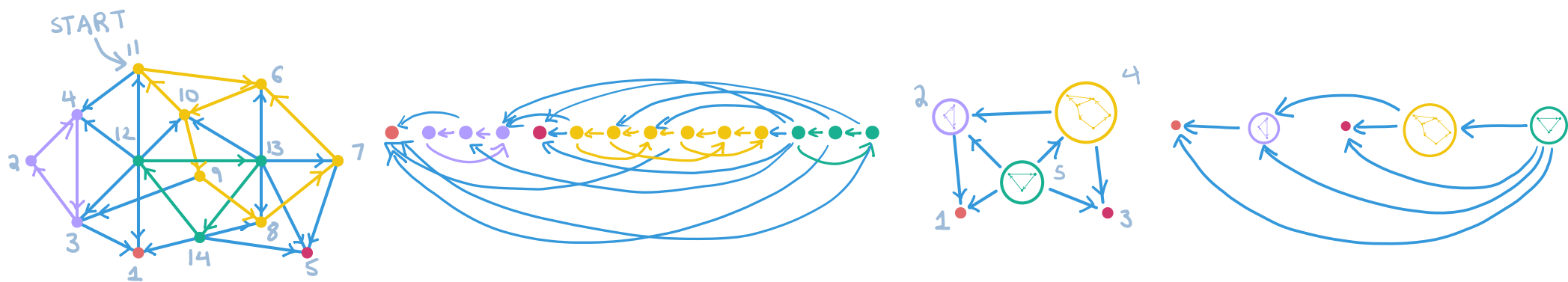
label each vertex in C w/
(some label for) C

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

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

Chapter 6

Searching and sorting graphs

Recall that graphs consist of vertices (denoted V) and edges (denoted E) that connect pairs of vertices. This chapter is exclusively about *directed graphs*, where edges are directed and represented by ordered pairs (u, v) . Here u and v are vertices, and the edge (u, v) is said to be directed from u to v . Directed edges are also known as arcs. For a directed edge/arc $a = (u, v)$, u is called the *tail* or the *source*, and v is called the *head* or the *target*, of a . A directed graph is allowed to have edges in both directions (u, v) and (v, u) between two pairs.

A *walk* is a sequence of arcs $e_1, \dots, e_k$ where the head/target of one arc e_i is the tail/source of the next arc e_{i+1} , for $i \in \{1, \dots, k-1\}$. The source of e_1 is called the starting point, and the target of e_k is called the end point or terminating point of the walk. For two vertices u and v , a *walk from u to v* is a walk starting at u and ending at v . We say that u can *reach* a vertex v if there is a walk from u to v .

Seemingly anything can be modeled as a graph. We give a few examples where the modeling is fairly straightforward. The world wide web is a directed graph. Every website is a node. Every link from one website to another is a directed edge from the first website to the second. Modern search engines are based on analyzing this graph.

Another example comes from programming. Modern programming languages allocate most of their data from a heap, where small blocks of memory are encapsulated by objects. These objects refer to each other by their locations in virtual memory. Here we have a graph where the objects are nodes and the pointers across objects are directed edges. In this model, *garbage collection* can be understood as a graph theoretic problem where the goal is to identify all the objects that cannot be reached by any sequence of pointers starting from a pointer on the stack, so that we can recycle the underlying memory.

Lastly, consider any road map. We can think of each intersection as a node, and the streets between the intersections as edges. Most streets will be undirected (or equivalently bidirected) edges since they allow traffic in both directions. One-way



We usually assume that graphs are given as a list of (possibly weighted) edges. With this list, one can build different data structures to make certain operations more convenient. Here we list the two most common ones. The first is to maintain, for each vertex $v \in V$, the list of all edges leaving v and (optionally) the list of all edges entering v . This is called the *adjacency list* representation. The second is to encode the edges in a large $V \times V$ matrix A (i.e., a two-dimensional array indexed by vertices V), as follows¹. For every edge (u, v) , we set $A[u, v] = 1$. Alternatively, if the edges

100

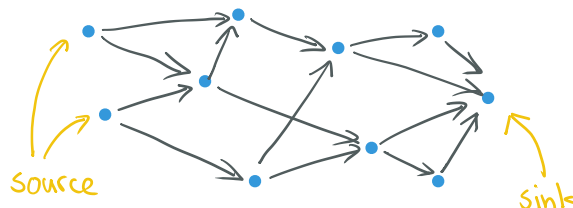
are weighted, then we set $A[u, v]$ to be the weight of the edge. The other coordinates corresponding to non-edges are left as 0.

There is an important functional difference between the adjacency matrix and the adjacency map. The adjacency list allows us to query, for each vertex v , the list of neighbors of v in $O(1)$ time per neighbor. The adjacency matrix does not, as one has to loop through the array to find each successive edge. On the other hand, the adjacency matrix allows us to query, for every pair of vertices $u, v \in V$, whether there is an edge from u to v , in $O(1)$ time. The adjacency list does not, since one has to scan the list of u 's neighbors to see if v is there.

Can one get the best of both worlds? Easily: just have both data structures on hand. One can also consider extensions such as using data structures (like search trees or hash maps) for each list of neighbors in the adjacency list data structure. We may discuss more advanced graph data structures later.

6.1 Sorting directed acyclic graphs²

A *cycle* is a walk that starts and ends at the same vertex. We start with an important special case of directed graphs where there are no directed cycles. Such a graph is called a *directed acyclic graph*, abbreviated *DAG*.



We say that a vertex $v \in V$ is a *source* if there are no edges directed into v , and a *sink* if there are no edges directed out of v . Every DAG has at least one source and at least one sink. (Why?)

Now, let $G = (V, E)$ be a DAG. Let v_1 be any source vertex in G . Observe that the graph G_1 obtained by removing v_1 from G (and all incident edges) is again a DAG. In particular, G_1 has at least one source vertex v_2 (which was either a source in G , or had only one incoming, which was from v_1). We can remove v_2 from G_1 to obtain a new DAG G_2 , and continue in the fashion, repeatedly removing vertices $v_3, v_4, v_5, \dots$

this is like randomly reindexing the vertices from 1 to $O(n)$.

²This section overlaps with section 9.3, which was about circuits, an important example of directed acyclic graphs. We repeat some parts about DAGS in general because we ran out of time to fully discuss circuits in lecture.

where for each i , v_i is a source in the graph G_{i-1} obtained by removing $v_1 \dots, v_{i-1}$ from G . The following subroutine encodes this process.

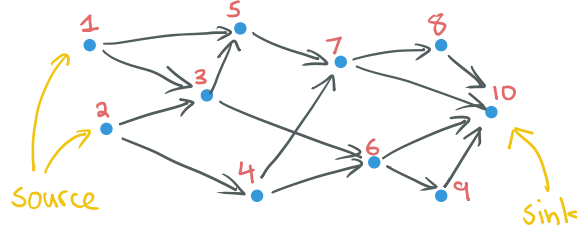
source-peeling($G = (V, E)$)

/ We assume G is a DAG. */*

**/*

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 .

Below we number the vertices of a DAG as they are peeled.



Consider the vertices $v_1, \dots, v_n$ in the order they are iteratively peeled from the graph. Let v_i and v_j be two vertices where v_i was peeled before v_j (i.e., $i < j$). When v_i was peeled, it was a source in a subgraph that included v_j . In particular, we cannot have a directed edge (v_j, v_i) from v_j to v_i . As this holds for all $i < j$, we see that all edges go from v_i to v_j .



The list output by the source-peeling algorithm is an example of a *topological order*. In general, an ordering $v_1, \dots, v_n$ of V is a *topological order* iff every directed edge in G goes from v_i to v_j for $i < j$. Equivalently, $v_1, \dots, v_n$ is a topological order if $i < j$ whenever v_i can reach v_j .

Topological orderings are extremely useful algorithmically. They give us a *sequential structure* of the graph on which we can apply the algorithmic techniques of chapter 5 (as we will see later). For these applications it is worth pointing out that **source-peeling** can also be made to run in linear time. Here one spends linear time to first calculate the in-degree³ of each vertex. The sources are the vertices

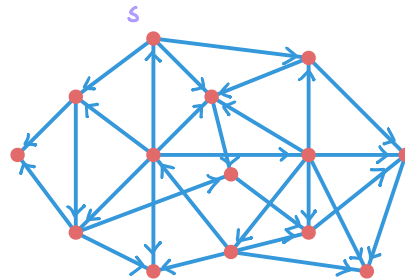
³The (unweighted) in-degree of a vertex v is the number of edges directed into v .

with in-degree 0. Whenever we peel a source v , for each arc (v, w) leaving v , we also decrease the in-degree of w as we delete that edge. We keep track of the vertices whose in-degrees are 0 so that the next source can be identified instantly.

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

6.2 Searching graphs

We consider the elementary problem of connectivity. Let $G = (V, E)$ be a directed graph with m edges and n vertices, and fix $s \in V$. We want to identify the set of vertices that s can reach. For example, consider the following directed graph. Can s reach all of the vertices in this graph?



Identifying the set of reachable vertices is a very practical computational task. Of course there is the literal interpretation of physical reachability on a map. Garbage collection, mentioned earlier, is also a matter of reachability. In planning problems, we interpret vertices as some kind of state, and edges as actions that take an entity from one state to another. Then reachability is asking whether it is possible to get from one initial state to another, desired state.

Let us first focus on the (s, t) -reachability problem. Here we have two vertices s and t and simply want to know if s can reach t . Consider the following natural recursive algorithm.

recursive-search(s, t)

/ Spec: Returns true if s can reach t , and false otherwise. */*

1. If $s = t$ then return **true**

/ See if any outgoing edge from s leads to t . */*

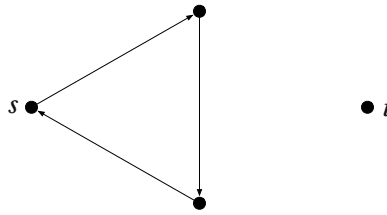
2. For each edge $(s, u) \in E$

A. If recursive-search(u, t) returns true then return **true**

3. Return **false**

The logic is clear. We are searching for t and are currently at s . If we are already on t then of course we are done. Otherwise, any path to t from s must start with an outgoing edge from s . So we try each of them and see if any of them will lead to s with a recursive call. If any succeed, then we return `true`. Otherwise, we return `false`.

There is a catch, however; more precisely, an *algorithmic* flaw in our argument. Consider, for example, the following graph.



The recursive search algorithm will spin around the triangle indefinitely. In general, any cycle in the graph will send `recursive-search` in a loop.

In general, when using induction to justify a recursion, we require some kind of ordering to avoid these “bad” cyclic recursive dependencies. That is why dynamic programming was so effective on sequences, where there are clear notions of “left and right” and we naturally steered clear of cyclic dependencies.

A helpful analogy is mazes. A maze is essentially an (s, t) -reachability problem. The entrance of the maze is s and the exit is t . When imagining a maze, though, keep in mind that the algorithm’s perspective is not the convenient birds-eye view that we see in puzzle books. Rather, we should think of ourselves as being *in* the maze, with tall walls to either side, and not having a clear sense of absolute location or direction. Imagine running through a maze, revisiting the same intersection again and again, feeling lost. That’s how `recursive-search` feels when it falls into a cycle.

6.2.1 Depth-first search

To solve mazes, let us take inspiration from folktales: breadcrumbs. Imagine being in a maze, or a dense forest. If we left breadcrumbs along our trail, we could keep track of where we’ve already been, and avoid a lot of wasted effort. Algorithmically, this is embodied by the simple idea of *marking*: whenever we visit a vertex v we mark it in memory. If it was already marked - meaning, we already visited v - then we stop exploring v and return (to wherever we were before visiting v). Otherwise we continue to explore v ’s edges. The following algorithm, called `depth-first-search` (and abbreviated DFS), augments `recursive-search` with marking.

DFS(v)

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

The algorithm is pretty simple. The following observation is obvious but it is helpful to make it explicit.

Observation 6.2. *If vertex v is marked, and $(v, w) \in E$, then w is (eventually) marked.*

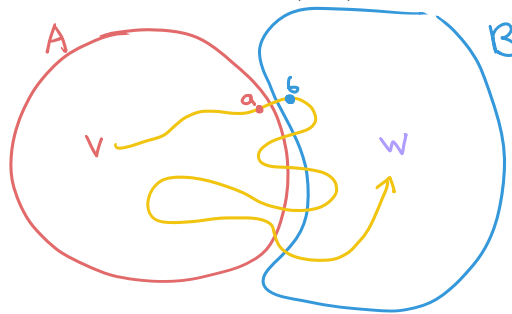
This sets up the main structural invariant of depth-first search.

Lemma 6.3. *When DFS(v) returns, all vertices reachable from v have been marked.*

Proof. Consider the moment that DFS(v) returns. Let

$$A = \{\text{marked vertices}\} \text{ and } B = \{\text{unmarked vertices}\}.$$

Let w be reachable from v , and suppose by contradiction that $w \in B$. Since w is reachable from v , there is a walk p from v to w . Since that path starts in A and ends in B , there must be a directed edge $(a, b) \in p$ such that $a \in A$ and $b \in B$.



But having a marked, b unmarked, and $(a, b) \in E$ contradicts observation 6.2. $\square$

Remark 6.4. Lemma 6.3 applies to recursive-search as well, except for that recursive-search may never terminate.

Now we analyze the running time.

Lemma 6.5. *DFS(v) runs in $O(m + n)$ time.*

Proof. We only traverse an edge $e = (v, w)$ after marking the initial point v . Each vertex gets marked once, so each is traversed at most once. $\square$

The two lemma's combine above give the following theorem about DFS.

Theorem 6.6. *In $O(m + n)$ time, $\text{DFS}(v)$ marks all the vertices reachable from v .*

So we can identify all vertices reachable from a vertex v in linear time.

6.3 Post-DFS Order

Having seen that $\text{DFS}(v)$ will search all vertices reachable from v , it is also interesting to study the *order* in which DFS searches the vertices.

In particular, consider the order in which DFS marks and finishes calls to $\text{DFS}(v)$. To make our discussion more concrete, let us modify DFS to print out vertices as we finish searching them.

$\text{DFS}(v)$

1. If v is unmarked:
 - A. Mark v .
 - B. For each edge (v, w) from v :
 1. $\text{DFS}(w)$.
- /* The following line is the only difference from the first version of
DFS. */*
- C. Append v to the output

In case one call to $\text{DFS}(v)$ does not search and print out all the vertices in the graph, we can wrap it in a simple driver that loops DFS over all v .

$\text{DFS-driver}(G = (V, E))$

1. For each vertex $v \in V$
 - A. $\text{DFS}(v)$

Ultimately DFS-driver prints out the vertices in some order $v_1, \dots, v_n$. We call any ordering $v_1, \dots, v_n$ of V produced by DFS-driver a *post-DFS ordering* of V . (“post” because each v is printed at the end of $\text{DFS}(v)$.) To understand this ordering, we first recall the basic invariant of DFS: $\text{DFS}(v)$ returns only when all vertices reachable from v have been marked. A slightly stronger observation is as follows.

Observation 6.7. *Consider a call to $\text{DFS}(v)$ for a vertex v . In the time between the beginning and end of the call, DFS only searches vertices that are reachable from v .*

This sets up the following lemma.

Lemma 6.8. *Let v, w be two vertices in a graph such that v can reach w but w cannot reach v . Then any post-DFS ordering lists w before v .*

Proof. We have two cases, depending on whether DFS marks v or w first. Suppose DFS visits marks v first. By lemma 6.3, DFS does not return from the recursive call $\text{DFS}(v)$ until all vertices reachable from v have been marked. In particular we call and complete $\text{DFS}(w)$ before the call to $\text{DFS}(v)$ completes.

In the second case, suppose w is marked first. Then the depth-first search will only explore vertices that w can reach before completing the call to $\text{DFS}(w)$ - and in particular, it will not search v before $\text{DFS}(w)$ terminates. $\square$

Now, suppose G is a DAG, and u can reach v . Then v cannot reach u because otherwise we would have a cycle. So v is always listed before u in the post-DFS order. If we apply this observation to every edge $(u, v) \in E$, we see that the edges are always directed up the list in the post-DFS ordering. That is, the post-DFS ordering gives a *reverse* topological ordering.

Lemma 6.9. *In $O(m + n)$, $\text{DFS-driver}(G = (V, E))$ outputs a list of the vertices V in reverse topological order.*

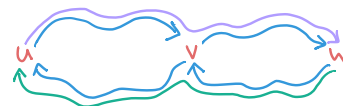
So DFS (plus reversing the list) gives a second way to topologically sort a directed acyclic graph.

6.4 Strongly-connected components

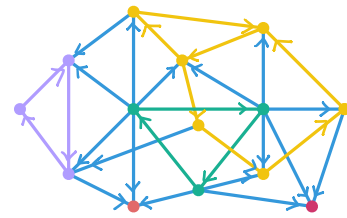
Two vertices $s, t \in V$ are *strongly connected* if s can reach t and t can reach s . A graph is strongly connected if every pair of vertices is strongly connected.

Lemma 6.10. *Strong connectivity defines an equivalence relationship. (In particular, if u and v are strongly connected, and v and w are strongly connected, then u and w are strongly connected.)*

We only sketch the proof as it is straight forward. Suppose u and v are strongly connected, and v and w are strongly connected. A path from u to v and a path from v to w can be concatenated to give a walk from u to w . Similarly we obtain a walk from w to u . So u and w are strongly connected.



Each maximal set of strongly connected vertices is called a *strongly connected component*. The strongly connected components partition V . On the right, we have colored the vertices of a graph by their strongly connected component.



We now consider the problem of computing all of the connected components in a directed graph G . It is easy to see that there is a polynomial time algorithm. For example, we can run $\text{DFS}(v)$ from every vertex v , which tells us for every vertex who they can reach. We can fill out a two-dimensional table with these results and identify all the strongly connected pairs. (This should take $O(mn)$ time, assuming $m \geq n$.) Can we do better?

Let us consider the simpler problem of identifying the strongly connected component of a particular vertex v . We can run $\text{DFS}(v)$ to identify all the vertices that v can reach. What is the fastest way to identify all the vertices that can reach v ?

We first mention a few reasonable ideas that have been suggested in lectures.

1. Run $\text{DFS}(u)$ from every vertex u visited by $\text{DFS}(v)$, and take the subset of vertices u that could reach v . This will work, but the running time is $O(n(m + n))$ which is not very impressive.
2. Run $\text{DFS}(u)$ only from the leaves u of the search tree carved out by $\text{DFS}(v)$. The intuition is that if u is a leaf of the search tree, and u can reach v , then all other vertices from v to u along the (u, v) -path in the search tree can also reach v . Therefore we do not need to test these intermediate vertices. We can also omit the vertices not visited by $\text{DFS}(v)$ since we already know that they are not strongly connected to v .

Again, this works, and while there may be fewer leaves than vertices overall, it is not true that in the worst case there are substantially fewer than n of them. So the worst case running time is still $O(n(m + n))$.

3. Another suggestion is to somehow adjust the $\text{DFS}(v)$ routine to try to detect back-edges and, somehow, detect the strongly connected components on the fly. I'm not sure if this can be made to work; certainly, it seems hard to prove it works.

Here is a helpful tip for this and many more problems: *don't change the algorithm; change the graph*. Consider the graph $G_r = (V, E_r)$ that has the same vertex set as

G but reverses the directions of all the edges. It takes only linear time to build G_r . Then a can reach b in G iff b can reach a in G_r . $\text{DFS}(v)$ in G_r will identify all the vertices that can reach v in G in $O(m + n)$. The strongly connected components of v is then the set of all vertices marked by $\text{DFS}(v)$ in both G and G_r . This algorithm takes only $O(m + n)$ time.

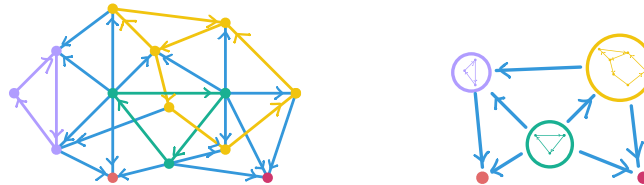
Thus we can find a single strongly connected component in $O(m + n)$ time. Let's move on to the greater task of identifying all the strongly connected components. A natural idea is to repeatedly use our clever $O(m + n)$ algorithm to identify and remove one strongly connected component at a time.

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

1. Until $V = \emptyset$
 - A. Let v be any vertex in V
 - B. Compute the strongly connected component C of v and add C to the output.
 - C. Remove C and all incident edges from G

Consider the running time of **SCC-peeling**. Potentially, we might remove only one vertex at a time. This gives a $O(n(m + n))$ running time which is hardly impressive.

Taking a step back, consider the graph G_C obtained by contracting each strongly connected component to a single vertex.



There is something conspicuous about this contracted graph: there are no directed cycles. This follows from maximality of the connected components. (If there was a cycle among the components in the contracted graphs, then all of these components would have to be strongly connected, a contradiction.) So the strongly connected components form a DAG. Consequently we can also speak about topological orderings of the DAG of SCC's, as well as sink SCC's and source SCC's.

Now, consider the peeling algorithm above, with the one modification that we always select a vertex from a sink component. Note that we don't know how to easily identify a vertex from a sink component quickly, but we will assume it for the moment and explore the consequences first.

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

1. Until $V = \emptyset$

/ We currently don't know how to identify the following vertex v . */*

A. let v be any vertex in a *sink component* of G

/ dfs(v) will search exactly the sink component of v . */*

B. $C \leftarrow$ all vertices marked by $\text{dfs}(v)$.

C. Append C to the output.

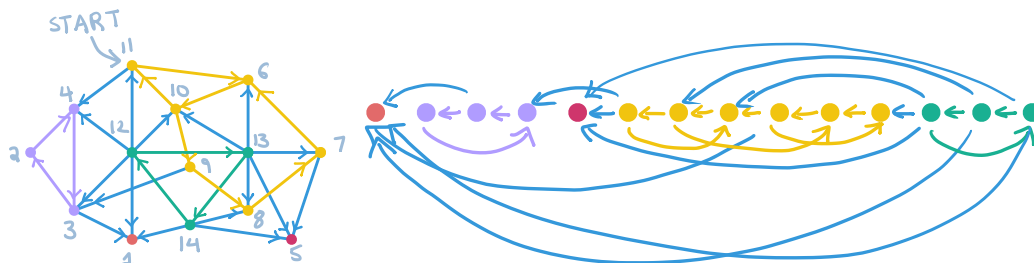
D. Remove C and all incident edges from G

Observe that if v is always selected from the sink component of the remaining graph, then $\text{dfs}(v)$ never leaves the sink component (by definition of sink). Thus, over the course of processing all the sink components, we only search each vertex and traverse each edge once. This gives a total running time of $O(m + n)$, a great improvement from $O(n(m + n))$. Of course, the algorithm is incomplete, because we won't know how to find vertices in a "sink-first" fashion. Thus the question remains: can we find a "sink-first" ordering of components?

Recall that depth-first search gives a (reversed) topological orderings in a DAG via the post-DFS order. More generally we proved (in lemma 6.8) that the post-DFS ordering has the following property in any directed graph:

If u can reach v but v cannot reach u , then v is listed before u . ($\star$)

This property does not tell us about the order between strongly connected pairs u and v . But suppose u and v are *not* in the same strongly connected component, and suppose there was a directed edge from u to v . Then u can reach v , and v cannot reach u , so v will be listed before u , and the edge (u, v) will be directed up the post-DFS order (so to speak). That is, if we list the vertices in post-DFS order from left to right, all of the edges that go *across* SCC's will be directed from right to left.



One might be tempted to declare that the first vertex must be from a sink component. (Which is suggested by the picture above.) Actually this is not quite true. But we can say something about the *last vertex* instead.

Lemma 6.11. *The last vertex in the post-DFS ordering is a vertex from a source component. More generally, given any ordering of the vertices satisfying $(\star)$ above, the last vertex is a vertex from a source component.*

Proof. Let $v_1, \dots, v_n$ satisfy $(\star)$; the claim is that v_n is in a source component. If not, then there exists some vertex v_i , where $i < n$, that can reach v_n , but v_n cannot reach v_i . But this contradicts $(\star)$. $\square$

Recall that we want vertices from sink components of G . We now have access to vertices from the source component. How do we get vertices from the sink component instead?

Don't change the algorithm; change the graph! Consider the reverse graph G_r . Sink component in G are source components in G_r and vice-versa.

Thus, finally, we can extract sinks components in the following fashion. We compute a post-DFS ordering $v_1, \dots, v_n$ in the reversed graph G_r ; we will fix that ordering for the rest of the algorithm. We take the last vertex v in this ordering, which belongs to a sink component in G . We find the connected component of v and remove it from G and G_r . Observe that the remaining ordering still satisfies $(\star)$. The last (remaining) vertex v belongs to a sink component in the remaining subgraph of G . Continuing in this fashion we keep extracting sink components from G and source components from G_r . The (corrected) pseudocode is as follows.

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

Thus we have shown the following.

Theorem 6.12. *Let $G = (V, E)$ be a directed graph with m edges and n vertices. In $O(m + n)$ time, `sink-first-SCC`(G) computes the strongly connected components of G . Moreover, it outputs a list of the strongly connected components in reverse topological order with respect to the DAG obtained by contracting the strongly connected components.*

The result is due to Tarjan [Tar72].

6.5 Additional notes and materials

This chapter overlaps with [Eri19, Chapters 5 and 6], [KT06, §3.1–3.3, 3.5–3.6], [DPV08, Chapter 3], [CLRS09, Chapter 22], and [DD11, (video) lecture 14].

Lecture materials. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)

Spring 2022 lecture materials. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

6.6 Exercises

Many more exercises related to the discussions in this chapter can be found in [Chapter 5](#) and [Chapter 6 in Jeff's notes](#).

Exercise 6.1. Prove that every directed acyclic graph has at least one source, and at least one sink.

Exercise 6.2. Prove or disprove: every walk in a directed acyclic graph has finite length.

Exercise 6.3. Consider the following recursive algorithm for exploring a directed graph with n vertices and m edges.

edge-search(v)

1. for each edge $e = (v \rightarrow w)$ leaving v
 - A. if e is unmarked
 1. mark e
 2. edge-search(w)

Analyze **edge-search** and give an upper bound (as tight as possible) on the running time, as a function of m and n .

Exercise 6.4. Let $G = (V, E)$ be a directed graph, and consider the problem of adding edges to G to make G strongly connected. The goal is to make G strongly connected by adding the minimum number of edges. Design and analyze an algorithm for this problem.

Exercise 6.5. Let $G = (V, E)$ be a directed graph. We say two vertices $s, t \in V$ are *half-connected* if either s can reach t or t can reach s . We say that the graph G is *half-connected* if every pair of vertices is half-connected. We say that two vertices s and t are *strictly half-connected* if either s can reach t , or t can reach s , but not both. We say that G is strictly half-connected if every pair of vertices is strictly half-connected.

1. Consider the problem of deciding if G is half-connected. Design and analyze an algorithm for this problem.
2. Consider the problem of deciding if G is strictly half-connected. Design and analyze an algorithm for this problem.

Exercise 6.6. Let $G = (V, E)$ be a directed graph with m edges and n vertices, such that there is at least one directed path between every pair of vertices (in at least one direction or the other). Suppose each vertex $v \in V$ is labeled by a distinct integer $\pi(v) \in [n]$ (that is, $\pi : V \rightarrow [n]$ is injective). We say a pair of vertices (u, v) is *reachably-misordered* if u can reach v and $\pi(v) < \pi(u)$.

Design and analyze an algorithm (as fast as possible) that counts the number of reachably-misordered pairs of vertices.

In section 6.2, we asked if a vertex s can reach all other vertices in a particular graph. The answer is no:

