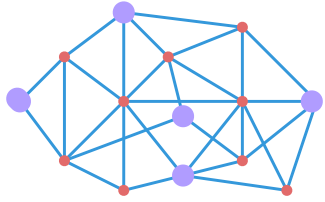


Searching and Sorting Graphs

Last time: Indep. Set in undirected graphs

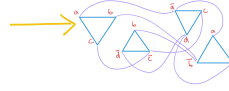
Input: undirected $G=(V,E)$

$S \subseteq V$ is independent if no $u,v \in S$ are connected by an edge.

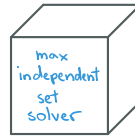


Goal: max cardinality independent set

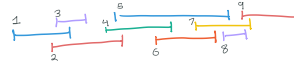
$$\phi(x_1, \dots, x_n) = (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3 \vee \neg x_4) \wedge \dots$$



yes, no

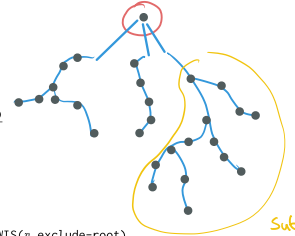


if ϕ is polytime, then entire algorithm is polytime.



MWIS-intervals($\mathcal{I} = \{I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]\}$, $w: \mathcal{I} \rightarrow \mathbb{R}_{\geq 0}$)
/* We assume $I_1 = [a_1, b_1], \dots, I_n = [a_n, b_n]$ is sorted in increasing order of a_i . */

1. If $n = 0$ then return 0
2. Let $i > 1$ be the first index such that $a_i > b_1$
// I_i is the first interval disjoint from I_1 , and $\{I_1, \dots, I_n\}$ is set of intervals disjoint from I_1
3. return the maximum of
 - A. MWIS-intervals($\{I_2, \dots, I_n\}$, w)
 - B. $w(I_1) + \text{MWIS-intervals}(\{I_i, \dots, I_n\}, w)$



MWIS(v , exclude-root)

1. If exclude-root
 - A. Return the sum of MWIS(x , false) over all children x of v .
/* Don't include v in the independent set */
2. Otherwise return the maximum of:
 - A. The sum of MWIS(x , false) over all children x of v .
/* Don't include v in the independent set */
 - B. The weight of v plus the sum of MWIS(x , true) over all children x of v .
/* Include v in the independent set */

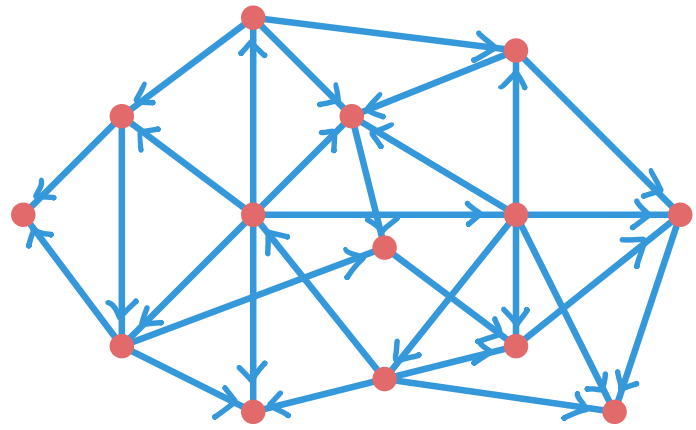
Today: directed graphs

acyclic graphs,

topological sort,

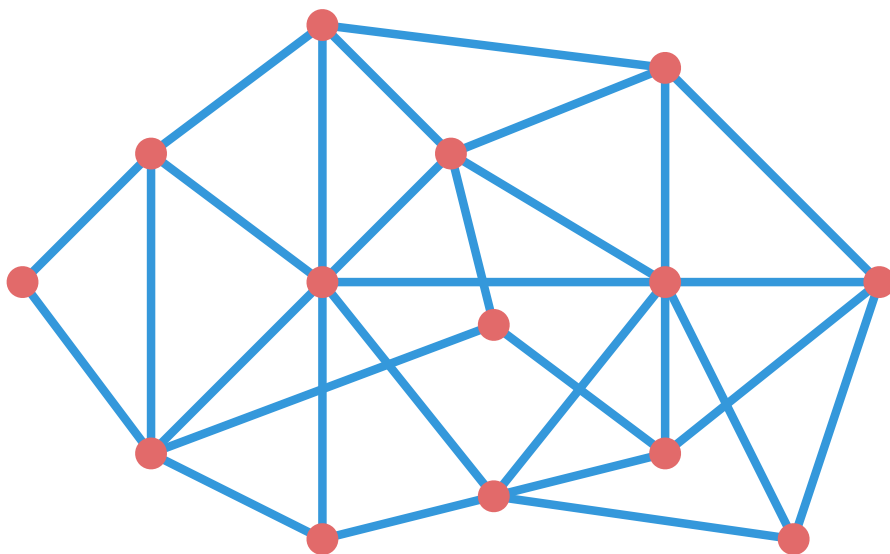
cycles, reachability, depth-first search,

strongly connected components



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

Graphs



vertices •

edges —————

$$G = (V, E)$$

$$n = |V|$$

$$m = |E|$$



undirected
edge
between u and v



directed
edge
from u to v

writing down graphs

"adjacency list"

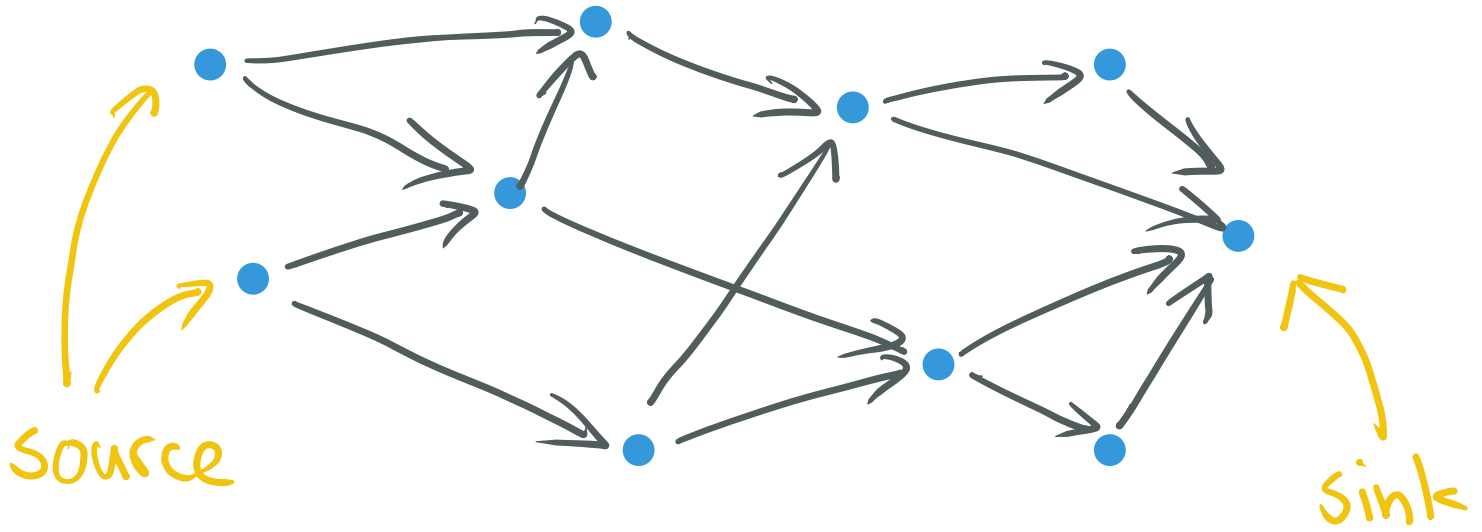
for each vertex, have a list of
all (incoming, outgoing) edges

e.g.

$v.out = [a, b, c, \dots]$

$v.in = [d, e, f, \dots]$

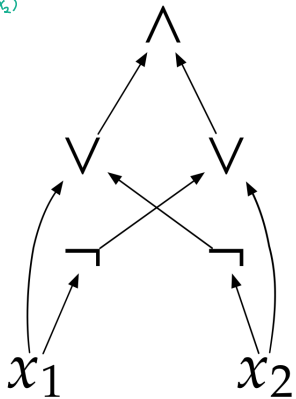
Directed Acyclic Graphs (DAG's) ^(no cycles)



e.g. circuits

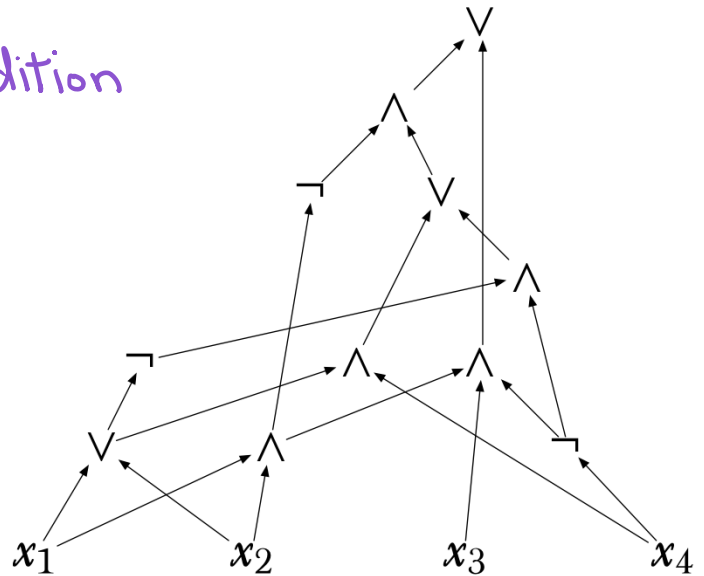
XOR ("exclusive-or")

(TRUE iff $x_1 \neq x_2$)



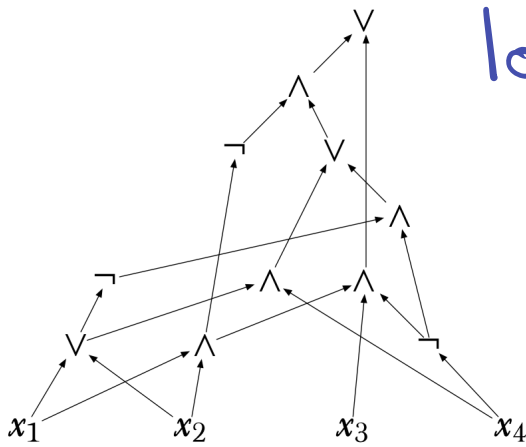
$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

Addition



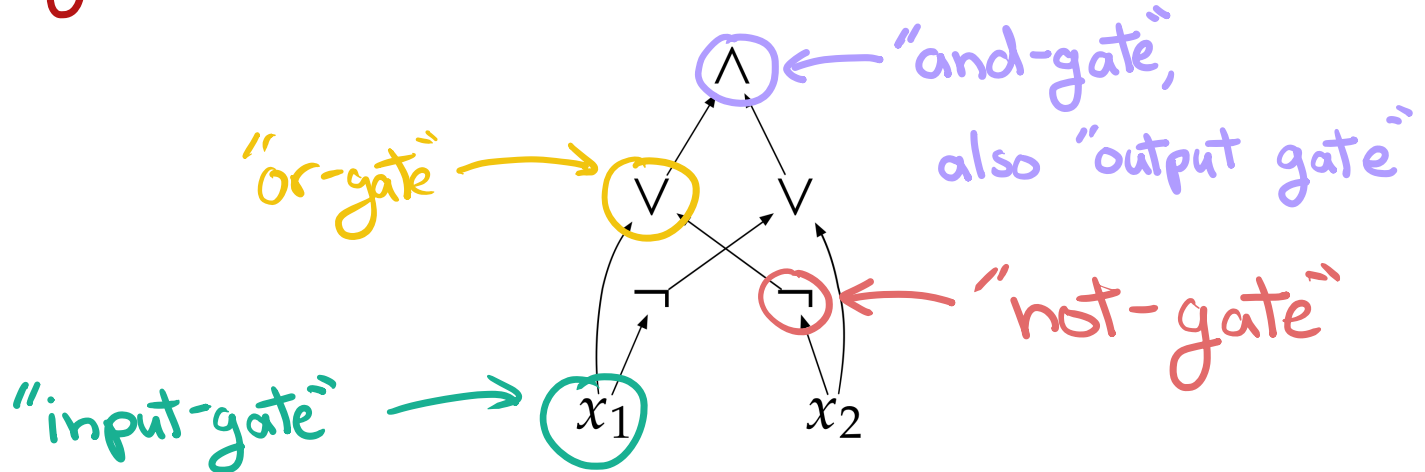
$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{if } x_1 + x_2 = x_3 x_4 \\ 0 & \text{otherwise} \end{cases}$$

(definition)

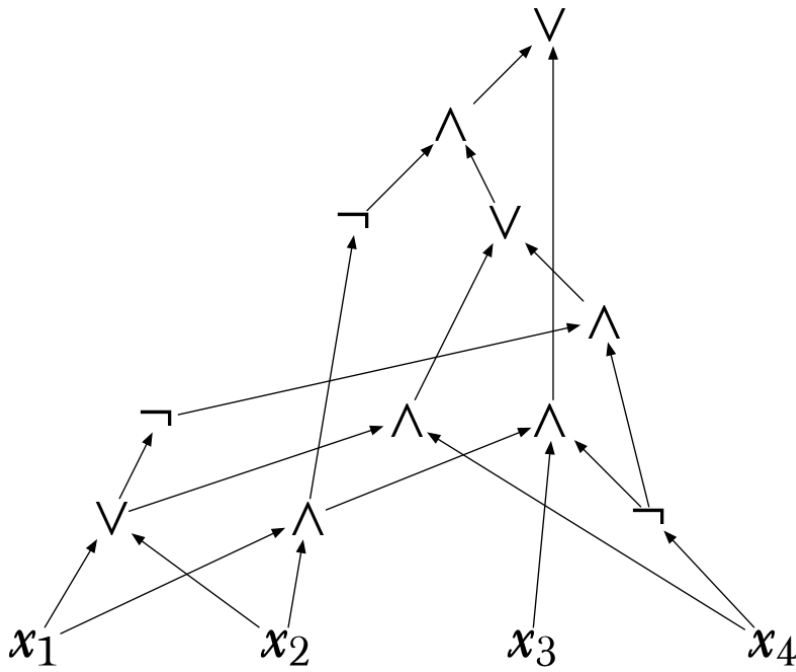


logical gates arranged
as vertices in a
directed acyclic graph

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



How to evaluate a circuit?

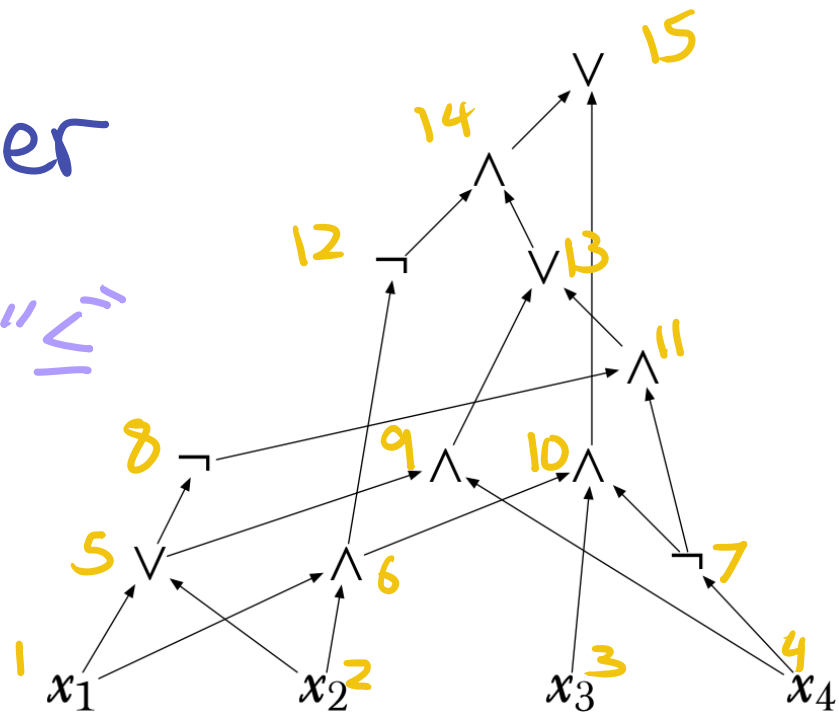
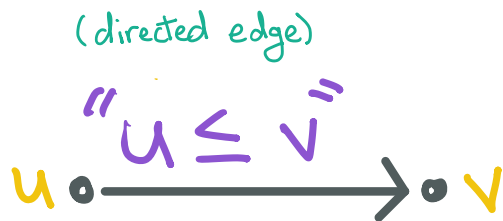


(conceptually)

"bottom up,"
one at a time

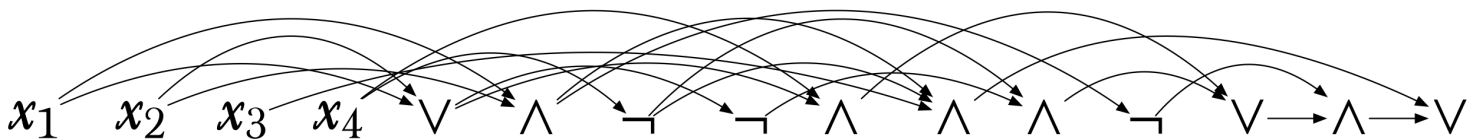
Topological order

think of arcs as " $\leq$ "



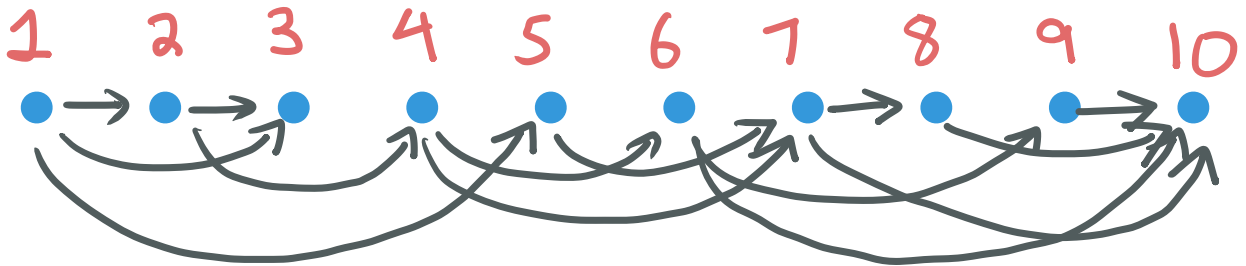
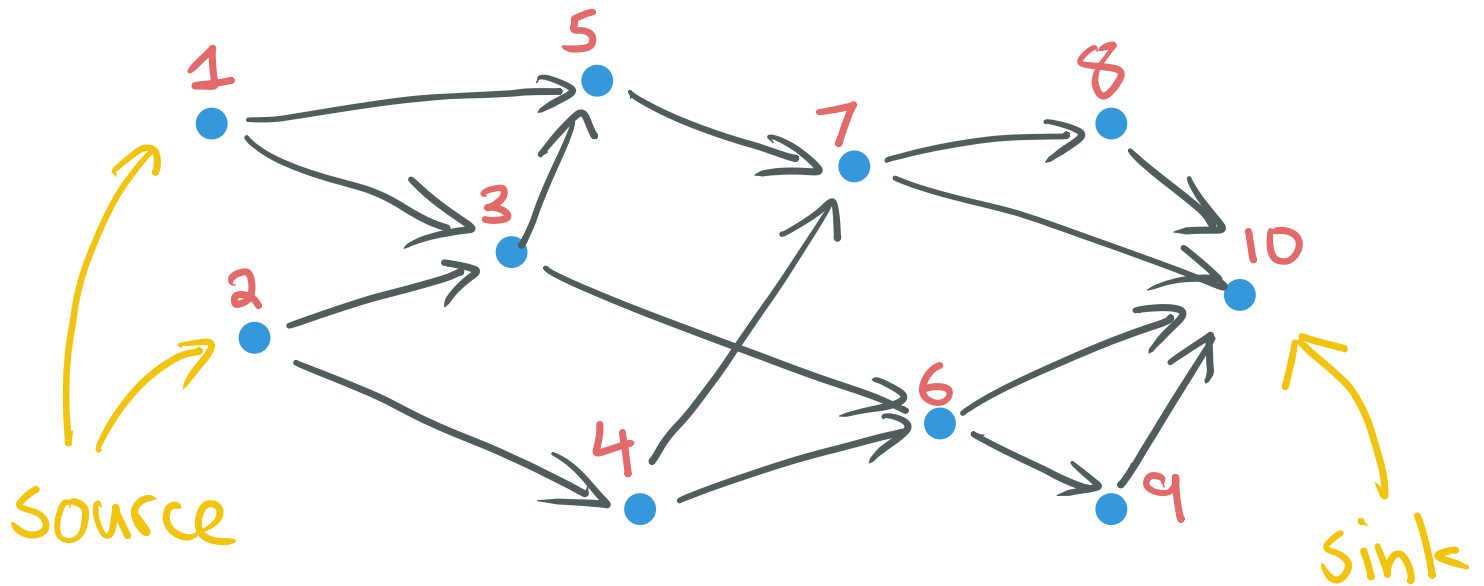
this defines a partial order on V

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



to evaluate, just process left to right!

Topological sort: keep removing sources

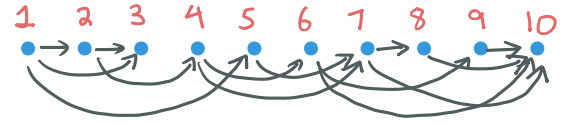
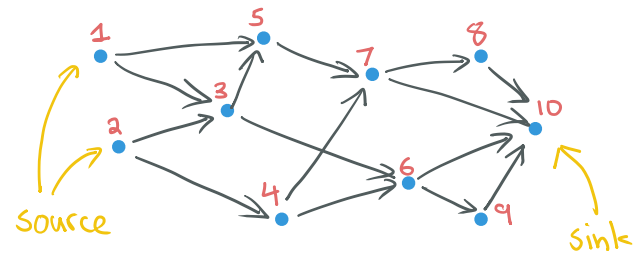


Note: topological order allows us to use "sequential brute force"

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

/ We assume G is a DAG.*

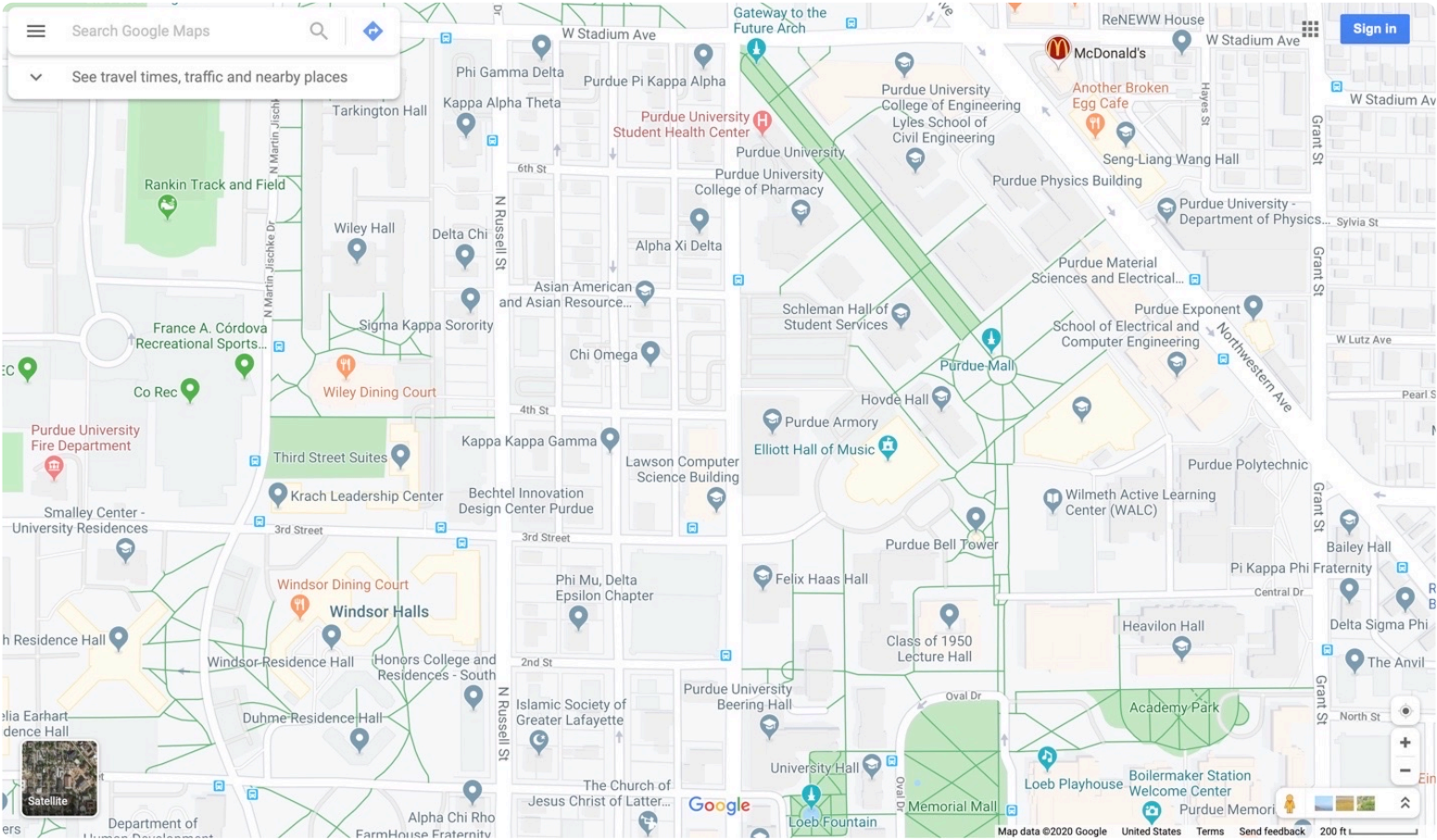
1. While there is a source $v \in V$
 - A. Append v to the output list.
 - B. Remove v and all incident edges from G .



↑ 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.*

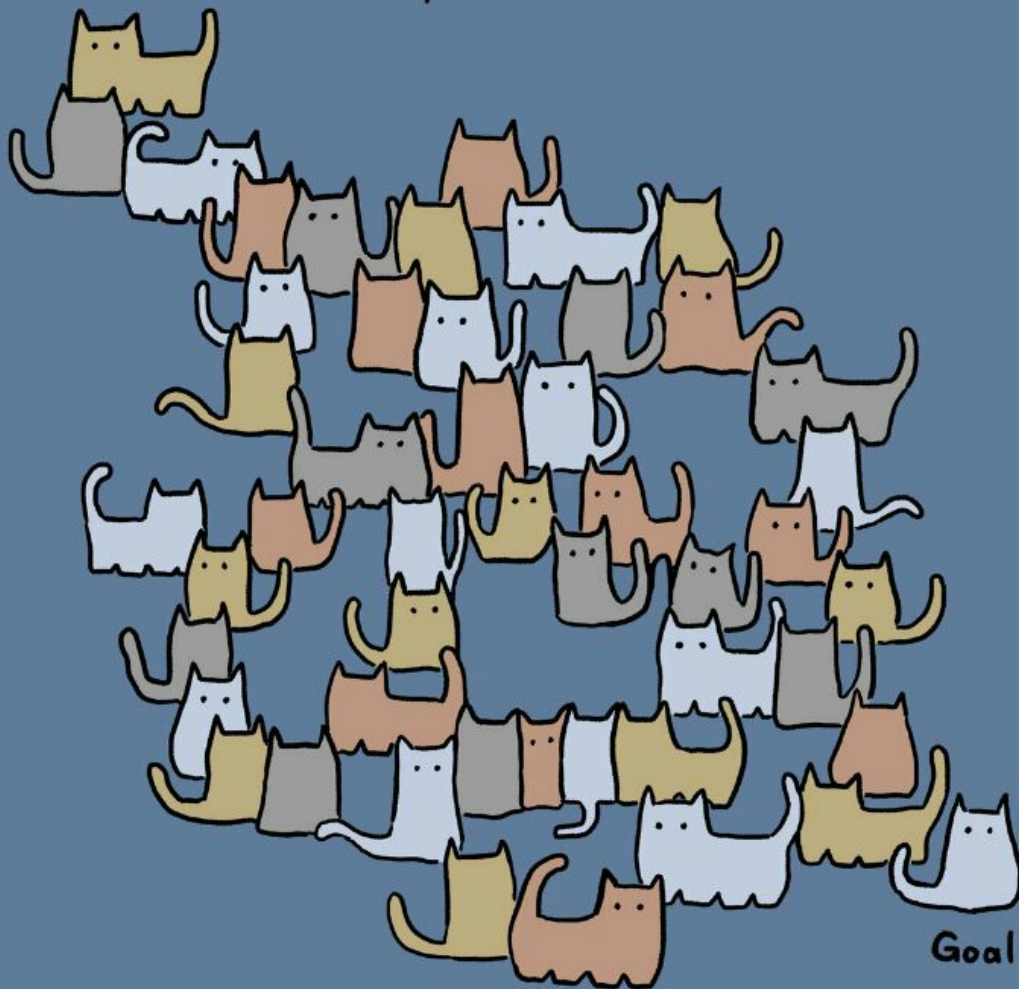
I Searching Graphs



CAT MAZE

Start

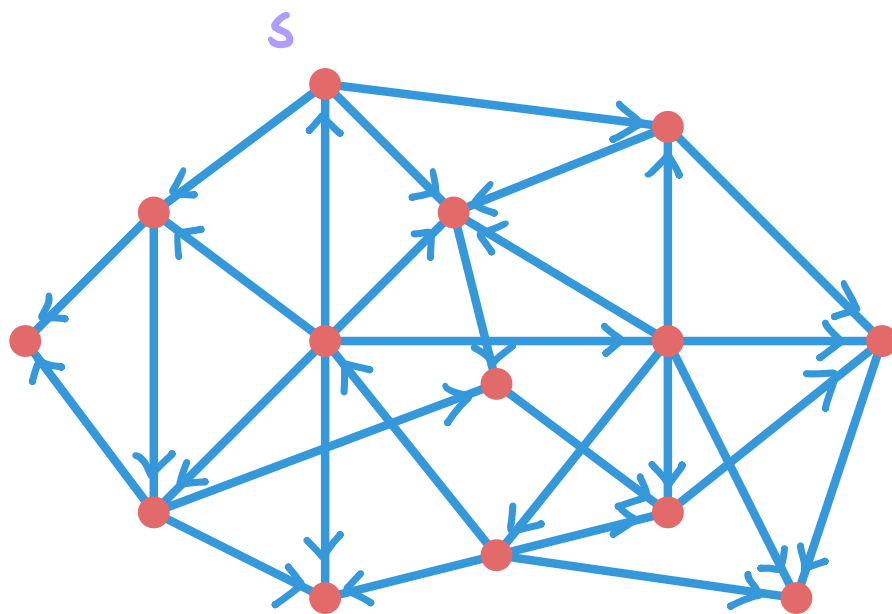
by Tatsunami



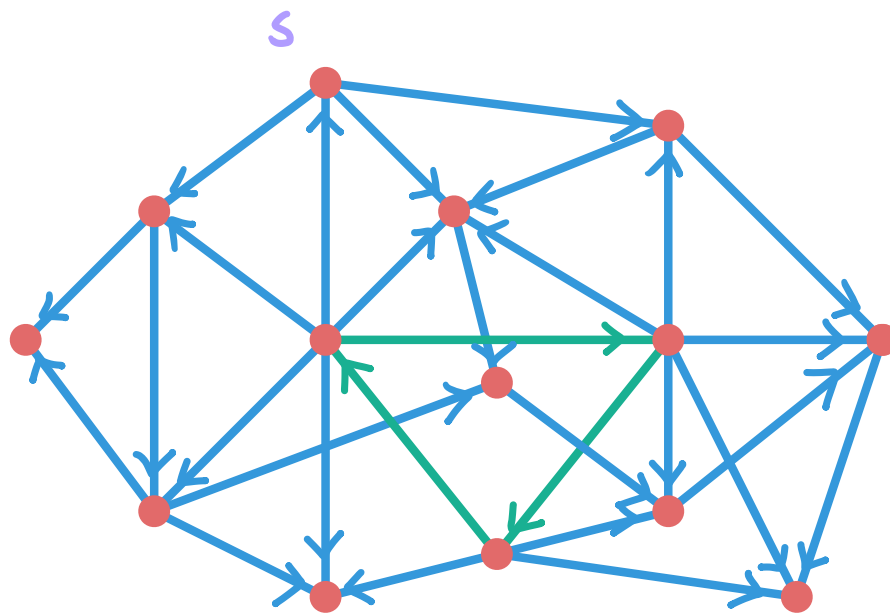
Reach the Goal by following the cats, alternating between cats facing the front and cats facing backwards.

The cats in your path must be connected or overlapping.

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



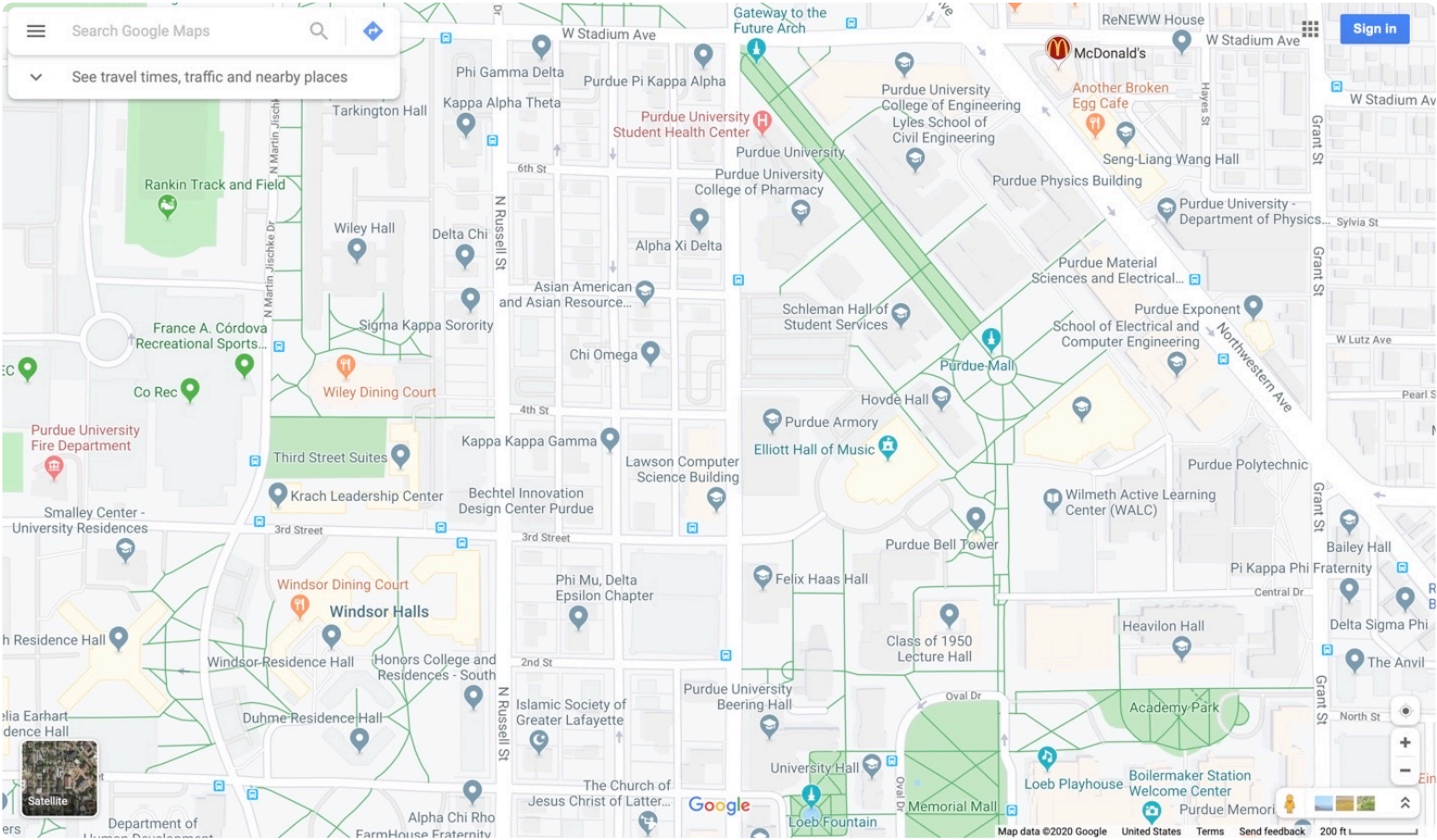
can s reach all of the other vertices?



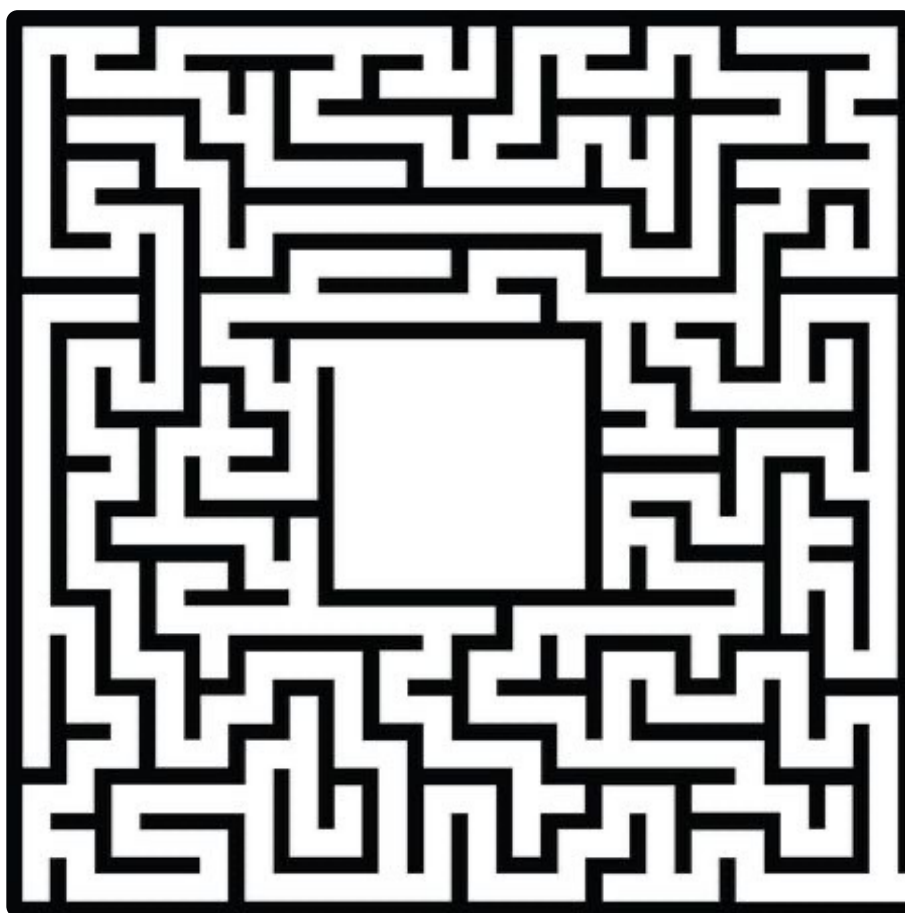
can s reach all of the other vertices?

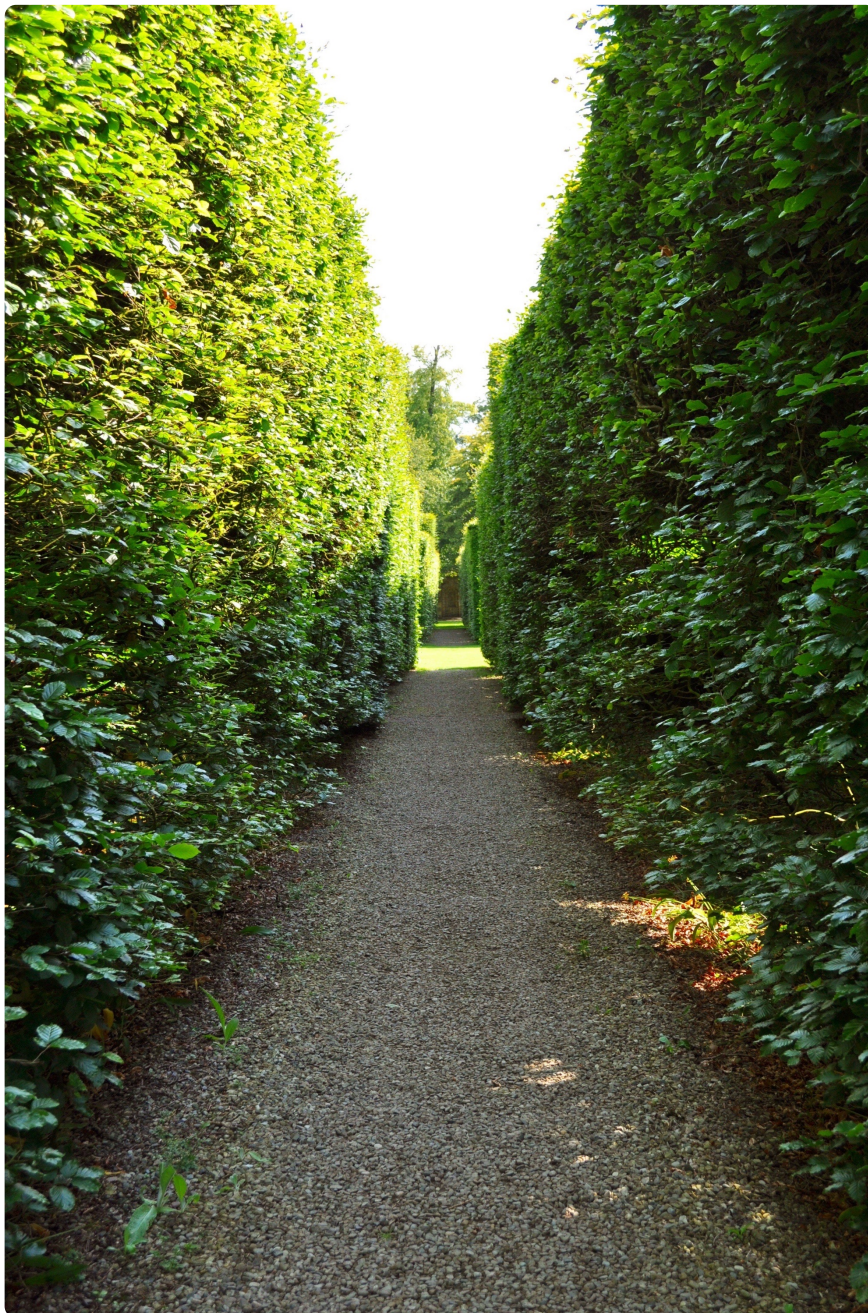
No

goal: design an algorithm to figure this out automatically



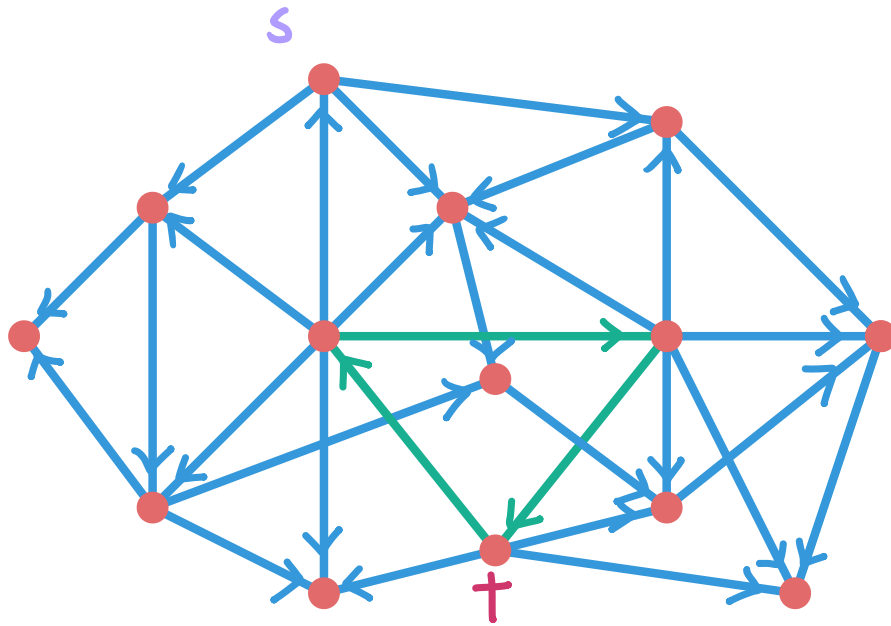






(harder w/out birds eye)

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



Goal: decide if s can reach t

Recursive-search(s, t)

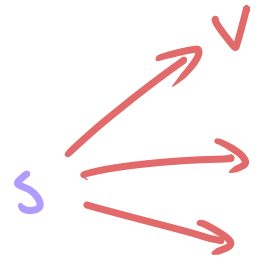
1. if $s=t$ then return TRUE

2. for each edge (s, v)

(a) if recursive-search(v, t) = TRUE

then return TRUE

3. return FALSE



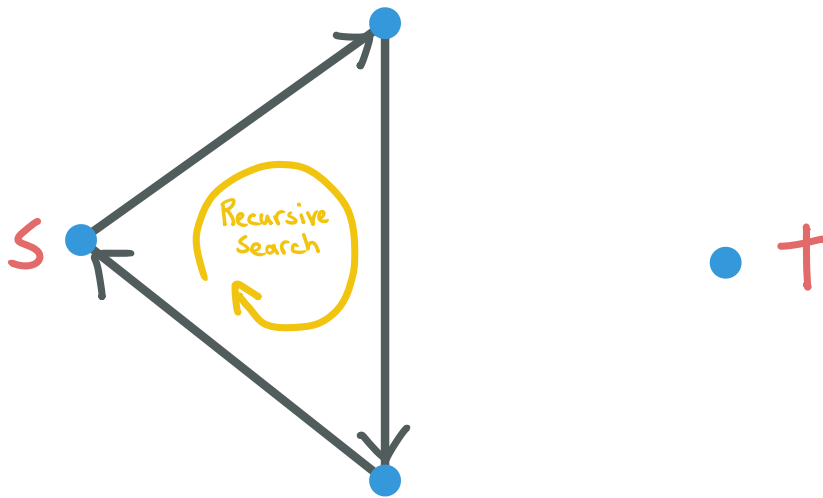
Argument

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

recursively $\text{Recursive-Search}(v, t)$ will
decide that v can reach t

Makes sense

But what about:



what went wrong?

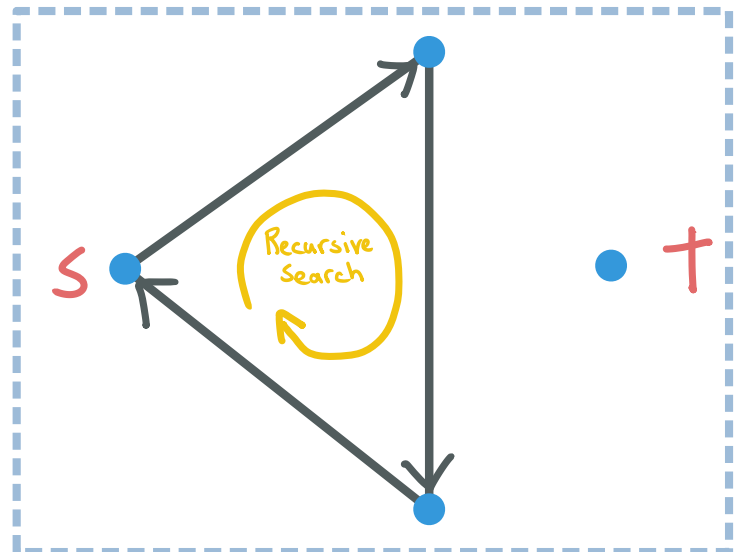
Argument

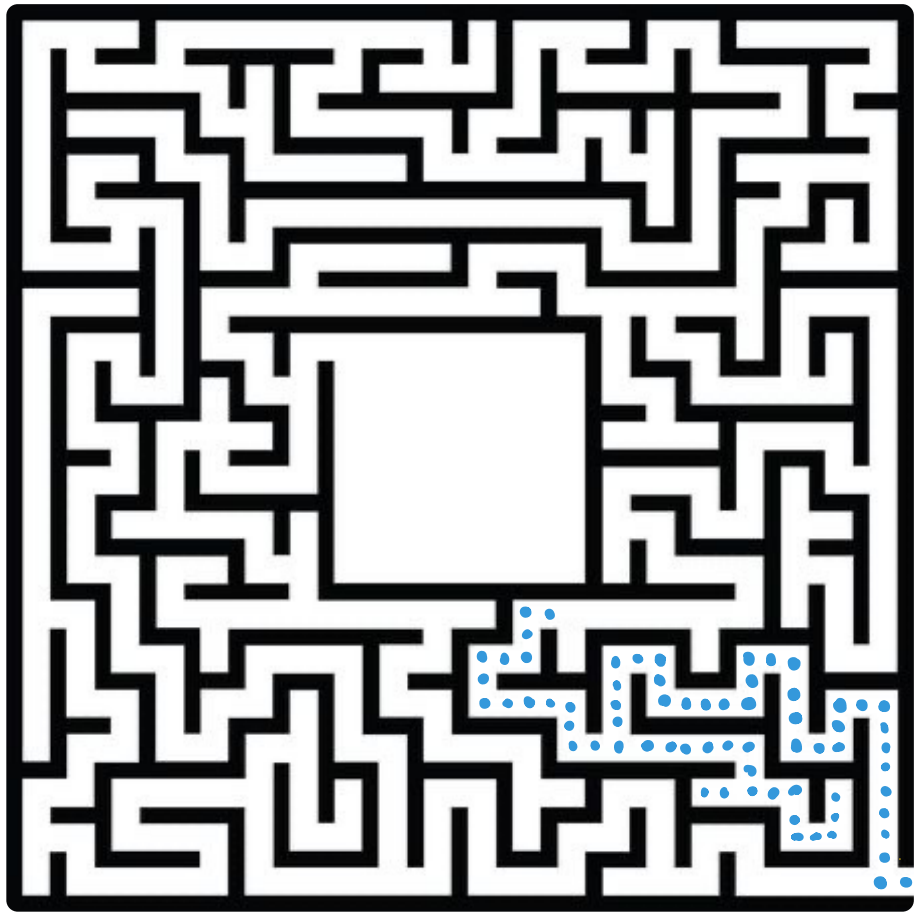
if s can reach t , and $s \neq t$, then
there must be edge (s, v) s.t.
 v can reach t

recursively $\text{Recursive-Search}(v, t)$ will
decide that v can reach t

recursion requires
induction to be
correct

cycles $\Rightarrow$ no clear
base case





Breadcrumbs

"depth-first search"

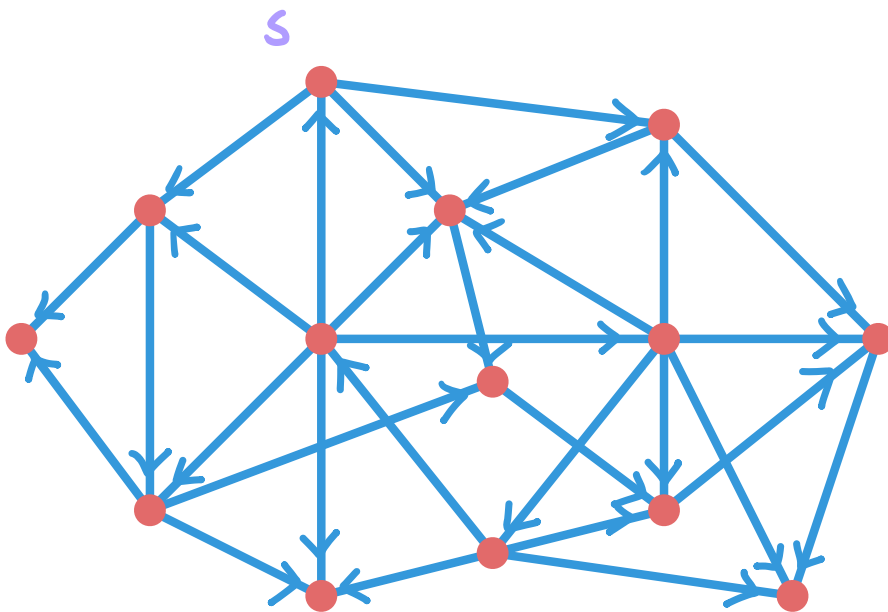
DFS(v):

① if v is unmarked

 Ⓐ mark v

 Ⓑ for each edge $v \rightarrow w$

 ① DFS(w)



can s reach all of the other vertices?

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

② DFS(v) visits all vertices reachable from v

DFS(v)

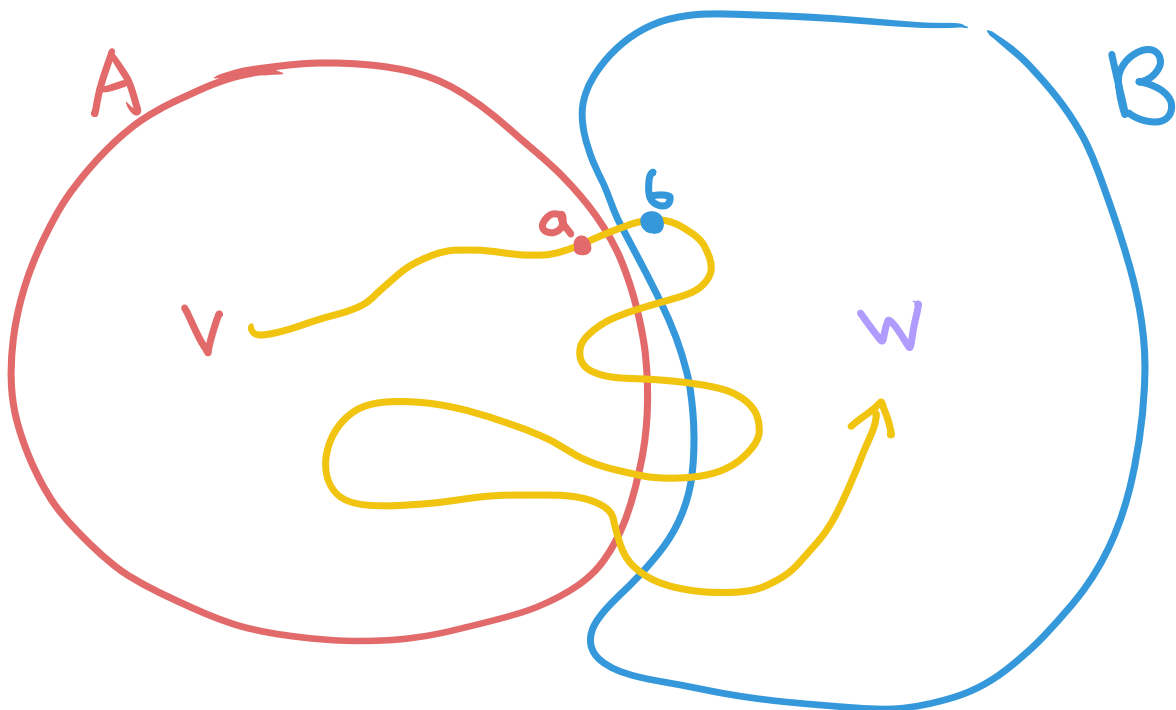
1. if v is unmarked
 - A. mark v
 - B. for each edge (v, w) from v
 1. DFS(w)

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

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

let w be reachable from v

suppose by contradiction that $w \in B$



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 $b \in B$

① DFS(v) runs in $O(m+n)$ time

DFS(v)

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

Proof

we traverse an edge $u \rightarrow v$
only after marking u

each vertex gets marked once

only traverse each edge once

"depth-first search"

DFS(v):

① if v is unmarked

 Ⓐ mark v

 Ⓑ for each edge $v \rightarrow w$

 ① DFS(w)

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

DFS is often used to visit all the vertices (maybe doing some processing at each vertex)

to make sure we visit all vertices,
use a "driver"/"wrapper"

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

while there is an unmarked vertex v

DFS(v)

Prologue: do we need breadcrumbs?

Suppose we only had $O(\log n)$ space
($\approx$ space for $O(1)$ vertices, counters)

undirected graphs w/ randomization:

take a random walk for $O(mn)$ steps

(taught in randomized algorithms)

undirected graphs, deterministically:

Algo. by Reingold 2008

(derandomizes the random walk approach,
also taught in randomized algorithms)

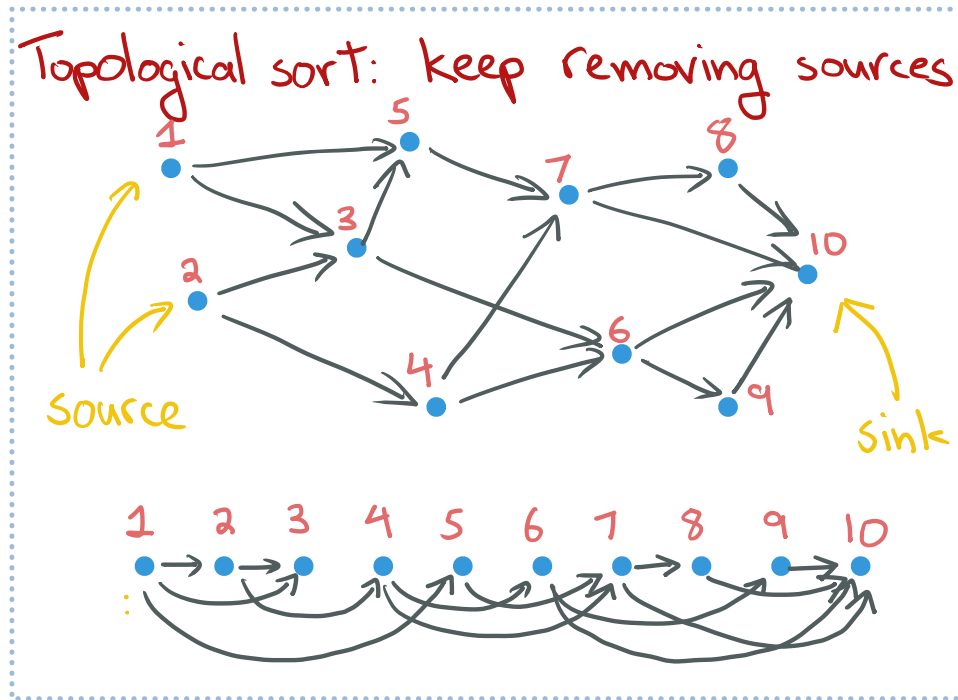
directed graphs, deterministically:

still not known!

Savitch 1970: $n^{O(\log n)}$ time,
 $O(\log^2 n)$ space

II Sorting Graphs

Recall:



Alternative approach based on DFS

Fact

consider $\text{DFS}(v)$

between time that

v is marked and $\text{DFS}(v)$ exits,

DFS only searches vertices reachable from v .

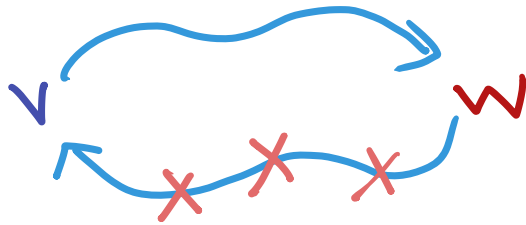
$\text{DFS}(v)$:

① if v is unmarked

② mark v

③ for each edge $v \rightarrow w$

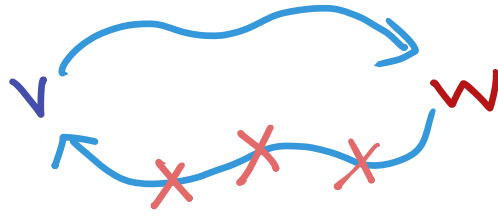
④ $\text{DFS}(w)$



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 $\text{DFS}(v)$



Proof

Case 1: DFS marks v first

then DFS marks w and

finishes DFS(w) before finishing w .

DFS(v):

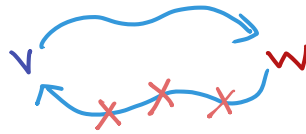
- ① if v is unmarked
- ② mark v
- ③ for each edge $v \rightarrow w$
 - ④ DFS(w)

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 $\text{DFS}(v)$

$\text{DFS}(v)$:

① if v is unmarked

 Ⓐ mark v

 Ⓑ for each edge $v \rightarrow w$

 ① $\text{DFS}(w)$

 Ⓒ print v

Consider DAG

DFS(v):

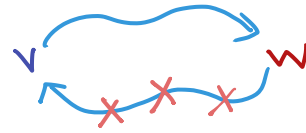
① if v is unmarked

 Ⓐ mark v

 Ⓑ for each edge $v \rightarrow w$

 ① DFS(w)

 ** Ⓒ print v



Lemma let $v, w \in V$ where

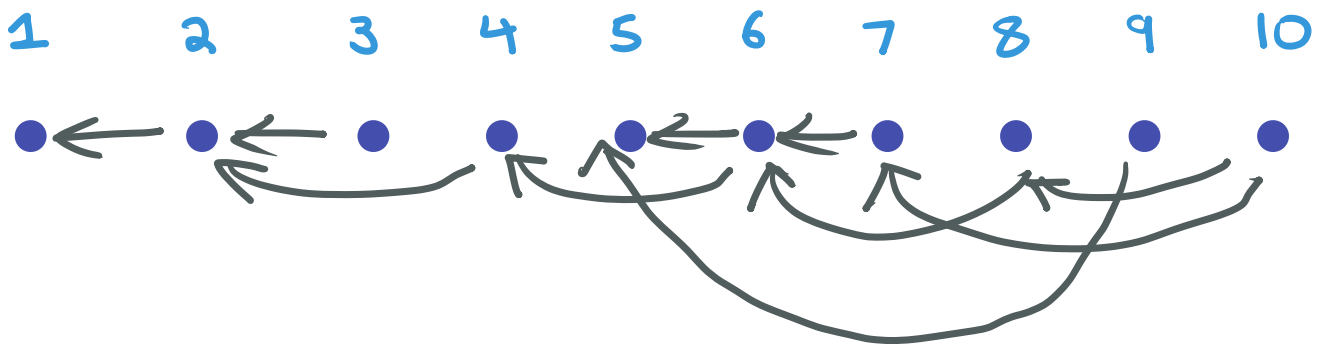
- v can reach w
- w cannot reach v .

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

if u can reach v ,

then v cannot reach u (why?)

so DFS prints v before u



all edges directed like 

Reverse topological order

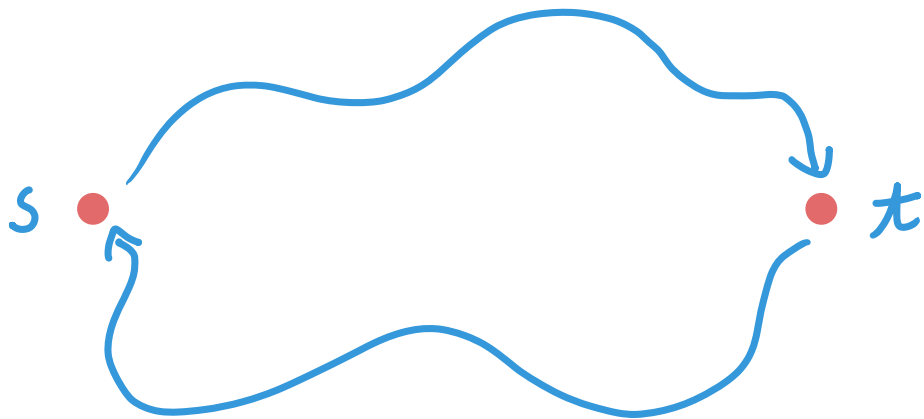
Recall:

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

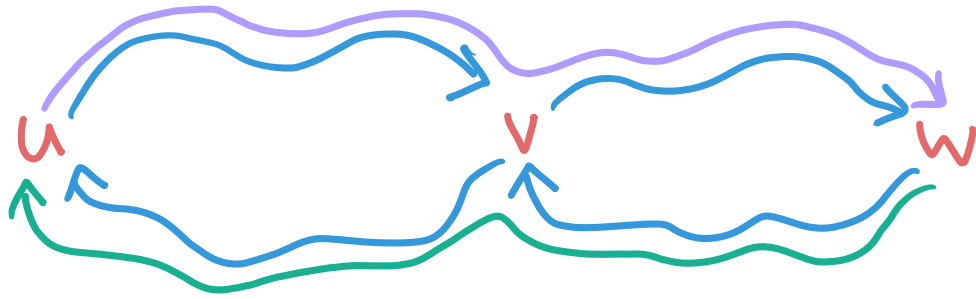


a graph is strongly connected if every pair of vertices is strongly connected

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)

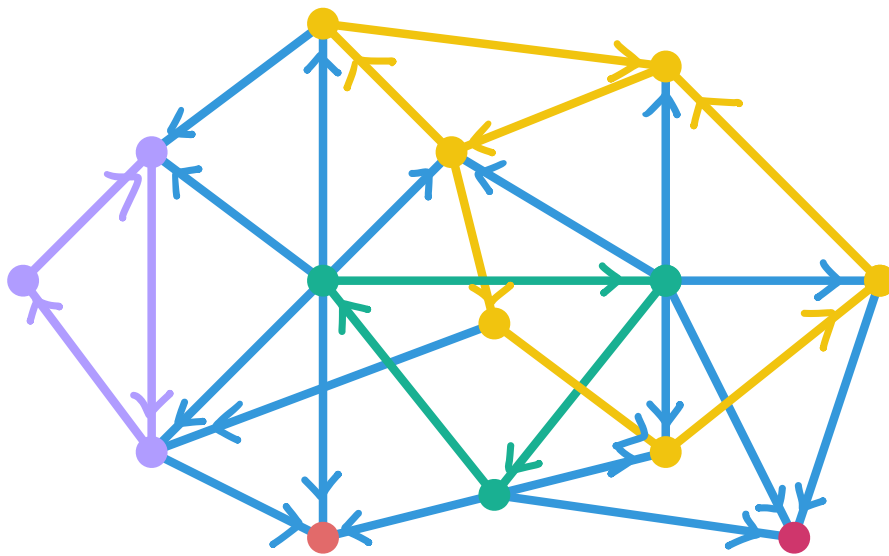
Proof



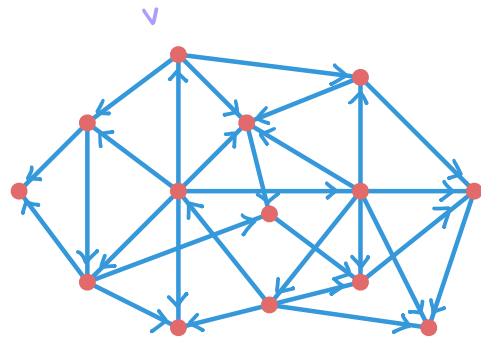
combine paths from $u \rightsquigarrow v$, $v \rightsquigarrow w$
to get path from $u \rightsquigarrow w$

similar for other direction

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



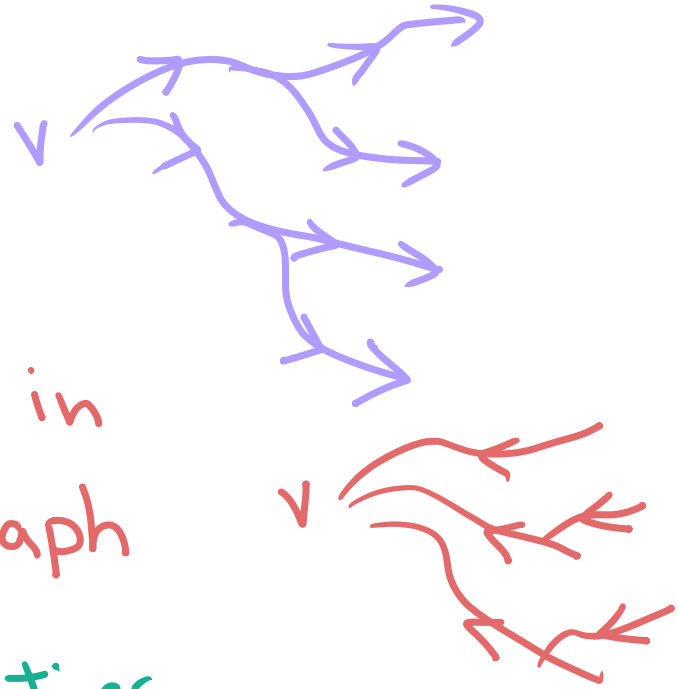
Goal: find the SCC's algorithmically
(strongly connected components)



First:

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

- 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)

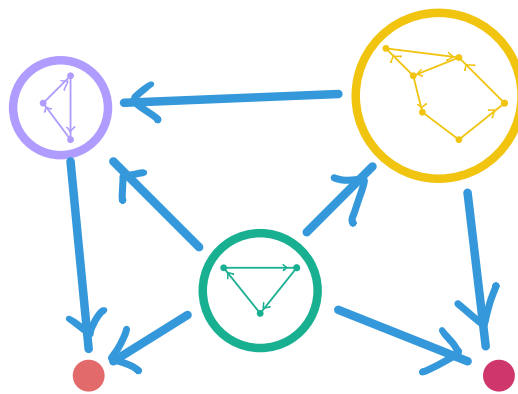
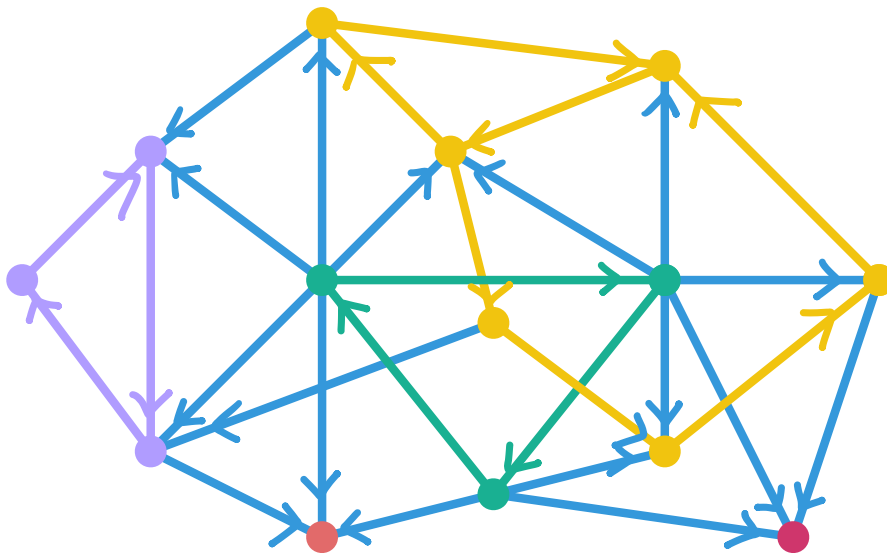
- ① run DFS from v $O(m+n)$
 - ② run DFS from v in the reversed graph $O(m+n)$
 - ③ return set of vertices visited in both DFS
-
- $O(m+n)$

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 ($G=(V,E)$)

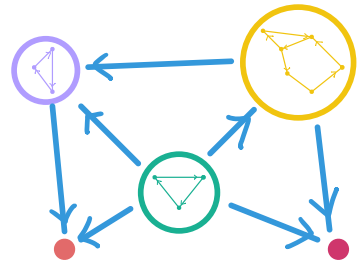
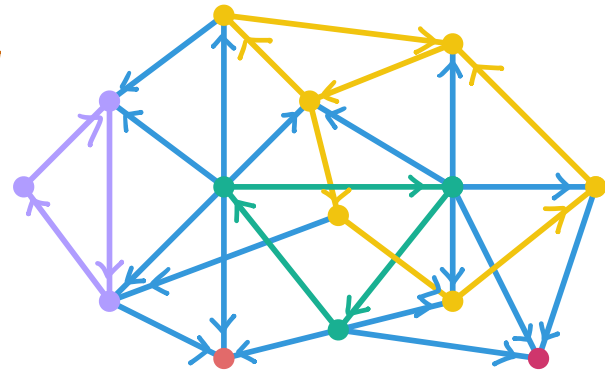
1. until $V=\emptyset$

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~~

DFS

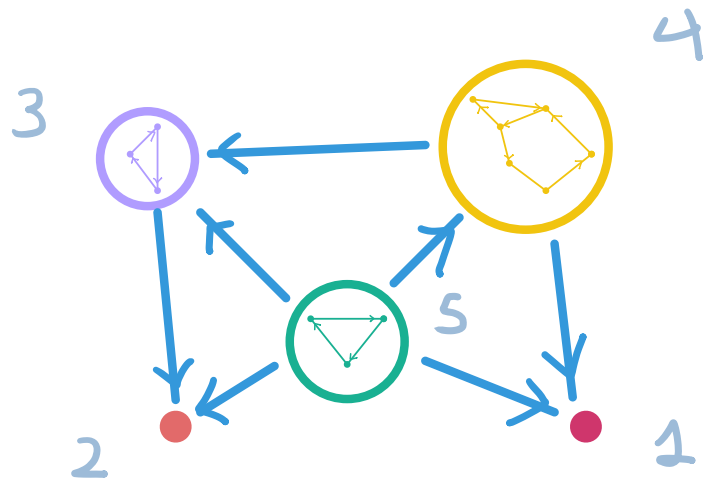
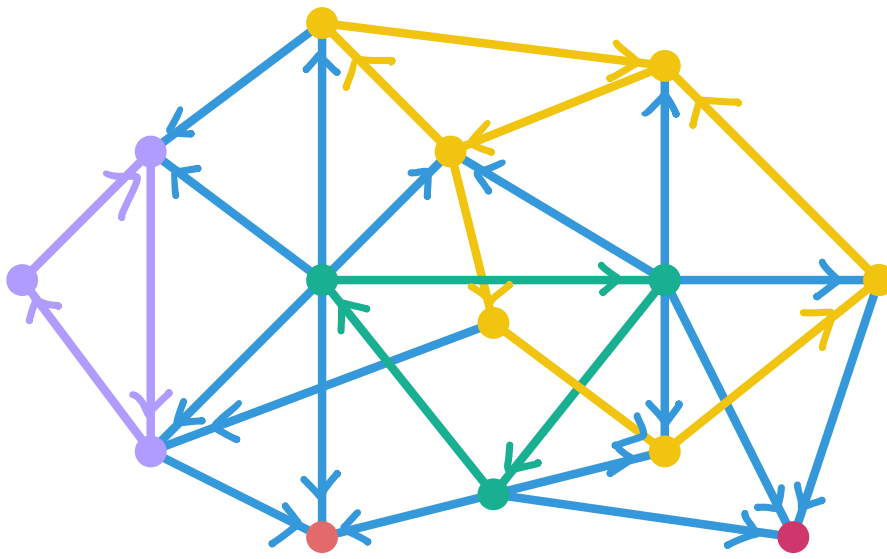
in a sink component

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

label each vertex in C w/ component

(some label for) C

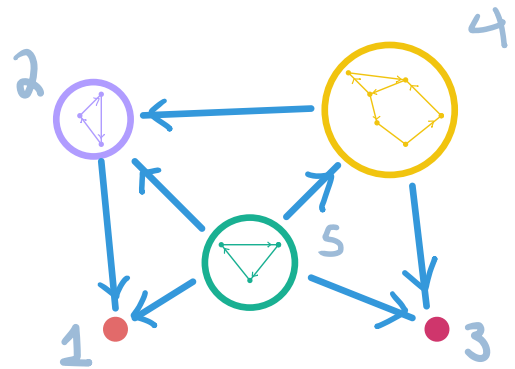
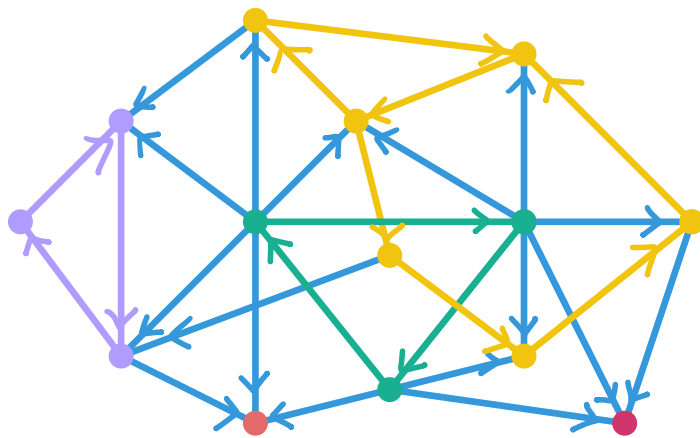
remove C and all incident
edges from G



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 SCC
(strongly connected component)

how to find "sink-component" ordering?

how to find "sink-component" ordering?

DFS!

DFS(v):

① if v is unmarked

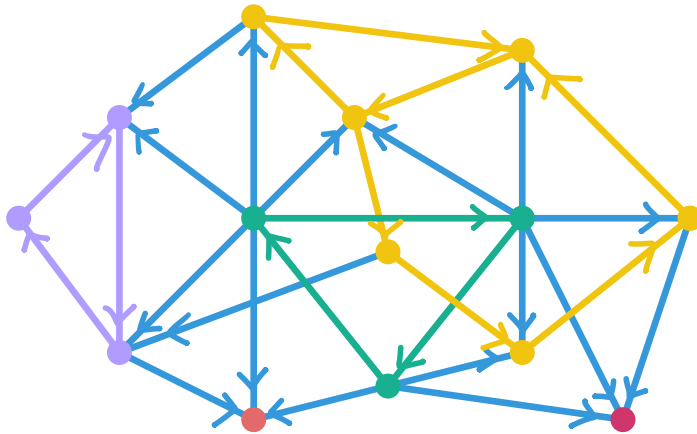
 Ⓐ mark v

 Ⓑ for each edge $v \rightarrow w$

 ① DFS(w)

★★ Ⓒ print v

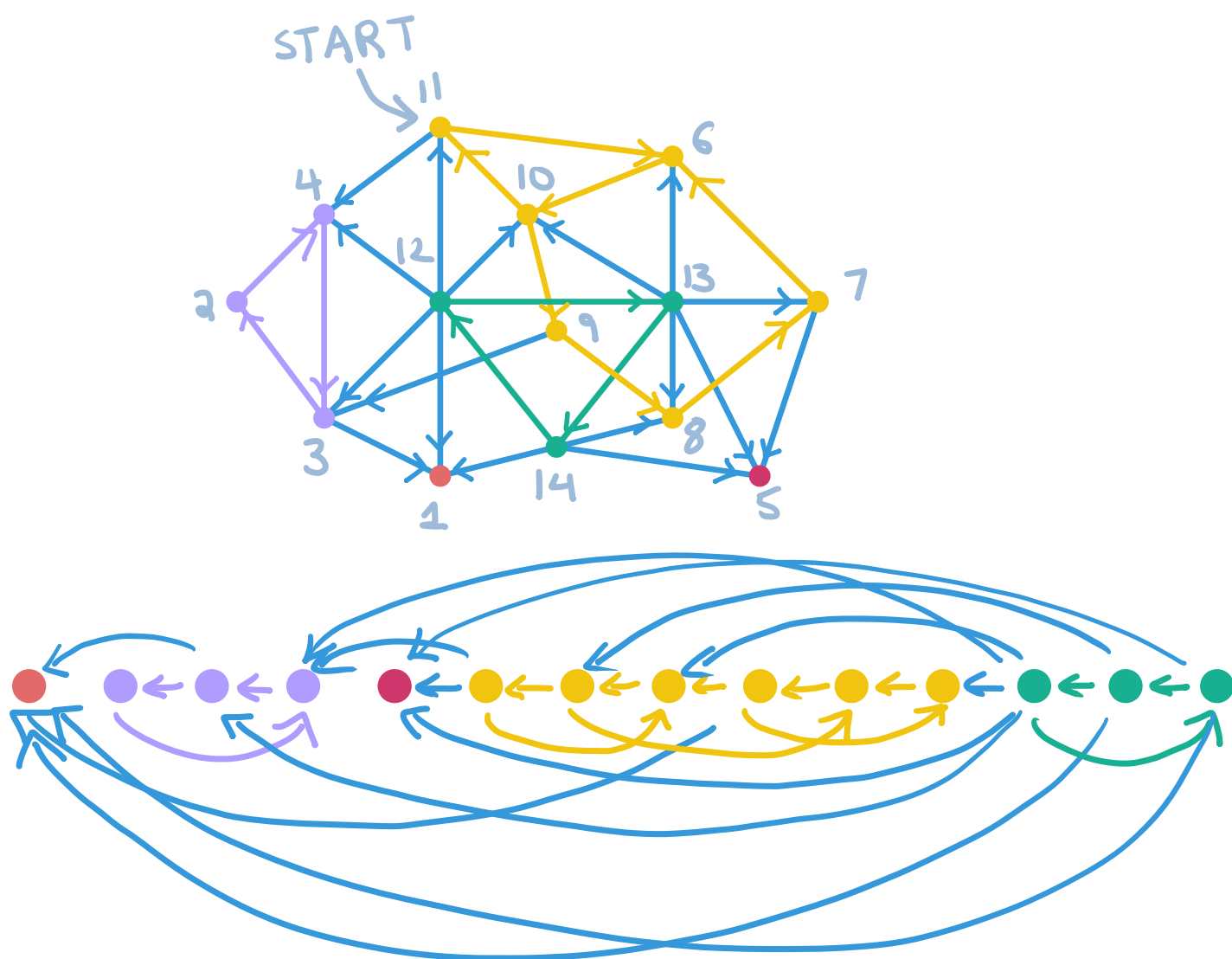
START



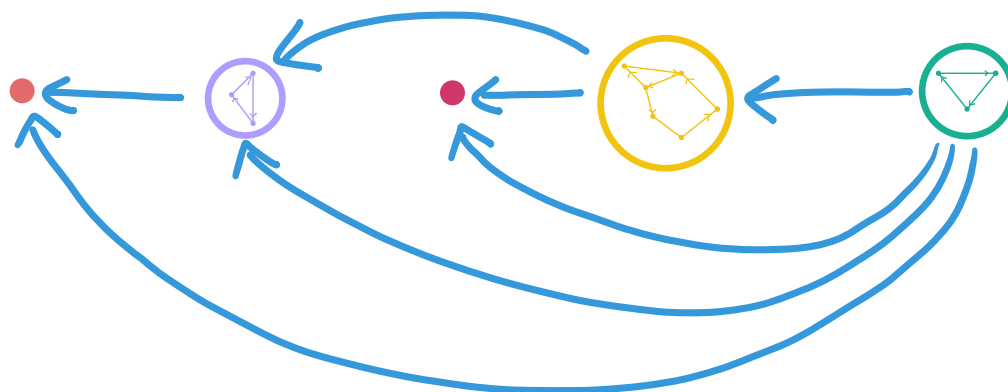
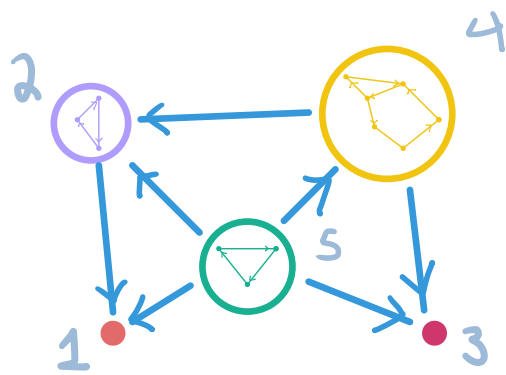
"post-order DFS traversal"

post-order DFS order

lists vertices in "sink-first order"



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



sink-first

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

until $V=\emptyset$

let v be ^{the next} ~~any~~ vertex in V

^{DFS} in post-DFS order

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

(some label for) C

remove C and all incident
edges from G

$O(m+n)$

[!!!]

Chapter 6

Searching and sorting graphs

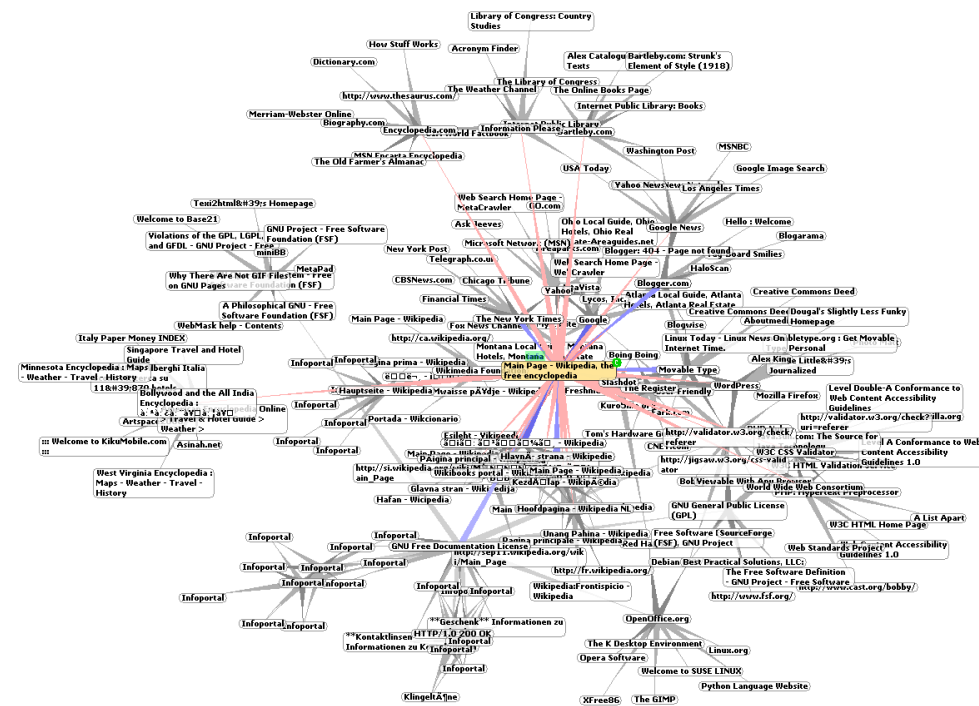
The previous discussion was about some optimization problems in undirected graphs; here we focus on directed graphs. The difference here is that the edges are *directed*, represented by ordered pairs (u, v) . Here u and v are vertices, and the edge (u, v) is said to be directed from u to v . A directed graph is allowed to have edges in both directions (u, v) and (v, u) between two pairs.

Seemingly anything can be modeled as a graph, but let us 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 streets are modeled by a directed edge in the direction of traffic. We can also annotate the edges with the speed limit and the length of the road, which gives us enough information to estimate how long it takes to go traverse that segment of the road.

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 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

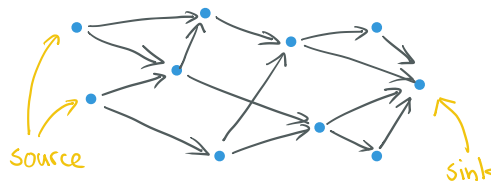
90

$u, v \in V$, 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

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 $v_3, v_4, v_5, \dots$ 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($G = (V, E)$)

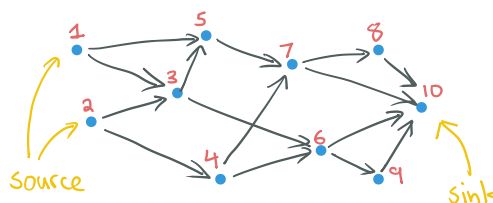
/ We assume G is a DAG.*

**/*

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.

²This section overlaps with section 2.3, 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.



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 $i < j$.

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 4 (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 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.

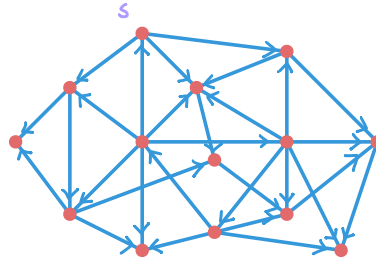
Theorem 6.1. *Given a DAG G with m edges and n vertices, one can compute a topological ordering of V in $O(m + n)$ time.*

6.2 Searching graphs

We consider the elementary problem of connectivity. Let $G = (V, E)$ with m edges and n vertices. A **walk** in a graph is a sequence of vertices along edges; i.e., a sequence $v_0, \dots, v_k \in V$ such that each (v_i, v_{i+1}) is a directed edge. A **path** is a walk that doesn't repeat vertices. We say that a vertex s can *reach* a vertex t if

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

there is a path from s to t in the graph. To test these definitions, consider the following directed graph. Can s reach all of the vertices in this graph?



Identifying the set of reachable vertices 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 from 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)

1. if $s = t$ then return true

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

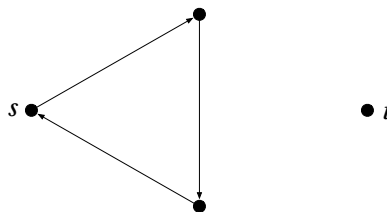
2. for each edge $(s, u) \in E$

A. if recursive-search(u, t) 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 the line, or on trees, where there are clear notions of “left and right”, or “top and bottom”, and we naturally steered clear of cycles.

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 can 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 left off before 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.*

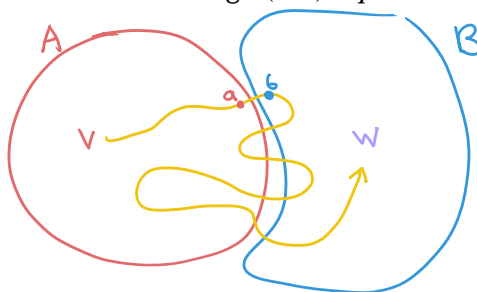
This sets up the main structural invariant of depth-first search.

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

Proof. Consider the moment that $\text{DFS}(v)$ returns. Let

$A = \{\text{marked vertices}\}$ 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 path 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. ■

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. $\text{DFS}(v)$ 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. ■

Combining our two lemma's in 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 Depth-first reverse topological sort

In section 6.1 we discussed directed acyclic graphs and how to topologically sort the vertices by iteratively peeling the vertices. It is also interesting to examine DFS when applied to DAG's.

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 (and applies to all graphs, not just DAGS).

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 (which again applies to all graphs).

Lemma 6.8. *Let v, w be two vertices in a graph such that v can reach w but w cannot reach v . The first call to $\text{DFS}(w)$ (marking w) exits before the first call to $\text{DFS}(v)$ (marking v) exits.*

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. ■

Suppose we run depth-first search on a graph and print out the vertices v as we complete each call to $\text{DFS}(v)$. The list is called a “post-DFS ordering”, since it is a post-order traversal. Let us make the process explicit in the following code.

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

By lemma 6.8, for every pair of vertices $u, v \in V$ such that u can reach v but v cannot reach u , v will be printed out before u . 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 sort.

Lemma 6.9. *In $O(m + n)$ time, post-DFS(v) outputs the vertices reachable from v in reverse topological order.*

If we want all the vertices in (reverse) topological order, and not just those reachable from a single vertex v , we can create a simple driver that loops post-DFS over all v .

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

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

Lemma 6.10. *In $O(m + n)$, post-DFS-driver($G = (V, E)$) outputs a list of the vertices V in reverse topological order.*

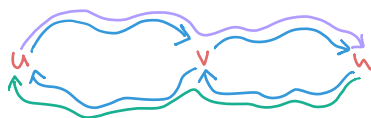
So DFS (plus reversing the list) gives a second way to topologically sort a directed acyclic graph.

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 6.11. *Strong connectivity defines an equivalence relationship (that is, if u and v are strongly connected, and v and w are strongly connected, then u and w are strongly connected).*

Rather than a full proof, we believe the following picture will suffice.



Each maximal set of strongly connected vertices is called a **strongly connected component**. The strongly connected components partition V . In the following pictures, we have colored the vertices of a graph by their strongly connected component. The strongly connected components can be computed,

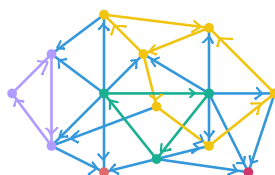


Figure 6.2: The strongly connected components of a graph.

rather crudely, by running dfs from every vertex and then examining the every pair of vertices if they could reach one another. To establish a base line, let us observe that this approach takes $O(n(m + n))$ time.

Let us consider the simpler problem of identifying the strongly connected component of a particular vertex v . We have a $O((m+n)n)$ time based on running dfs from every vertex. This seems excessive - can one do better?

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. Then a can reach b in G iff b can reach a in G_r . If we run $\text{dfs}(v)$ in G_r , then we 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 only takes $O(m+n)$ time.

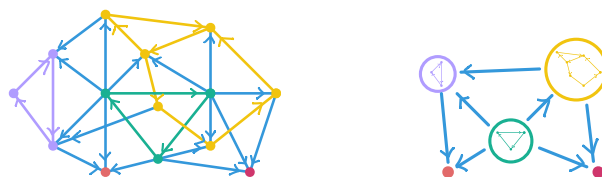
Thus we can find a single strongly connected component in $O(m+n)$ time. Let's move on to the greater task of identifying all the strongly connected components. A natural idea is to repeatedly use our clever $O(m+n)$ algorithm to identify and remove one strongly connected component at a time.

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

1. until $V = \emptyset$
 - A. let v be an 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. Since the strongly connected components form a DAG, we can also speak about topological orderings of DAG's, as well as sink components and source components.

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 $V = \emptyset$

/* At the moment, we don't know how to identify the following vertex
 v . */

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

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

B. $C \leftarrow$ all vertices marked by 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 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 running time of $O(m + n)$, a great improvement from $O(m(m + n))$. 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 our discussion from topological sort, where in particular we showed that depth-first search can also give a (reversed) topological orderings. In particular, we observed that for any two vertices $u, v \in V$ where u can reach v but not vice-versa – so in particular, u and v are not strongly connected – v will be listed earlier than u in any post-DFS ordering.

This means that, if we list the vertices in post-DFS order from left to right, all of the edges that go *across* components will be directed from right to left. One might be tempted to declare that the first vertex must be from a sink component. Actually this is not quite true. But we can say something about the *last vertex* instead.

Lemma 6.12. *The last vertex in the post-DFS ordering is a vertex from the source component.*

Proof. Let $v_1, \dots, v_n$ list the edges in post-order traversal; the claim is that v_n is in a sink component. If not, then there exists some vertex v_i , where $i < n$, that can reach v_n . But this contradicts lemma 6.8. ■

We want the sink component of a graph G , and we have access to the source component. How do we get the sink component instead? It's a trick we've used before: the reverse graph G_r . Any sink component in G is a source component in G_r .

Lemma 6.13. *Let G be a directed graph. The last vertex in the post-DFS order in the reversed graph G_r is a vertex from a sink component in the (original) graph G .*

Thus, finally, we can extract sinks components in the following fashion. We compute a post-DFS ordering in the reversed graph G_r . 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 . We still have a post-DFS ordering in the remainder of the reversed graph, where the (new) last vertex v belongs to a sink component in the remaining part of G . And 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 6.14. *Let $G = (V, E)$ be a directed graph with m edges and n vertices. In $O(m + n)$ time, $\text{sink-first-SCC}(G)$ 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.*

6.5 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 patriotic directed graph where all the edges are colored either red, white, or blue. An **American walk** is a walk where the edges alternate in color in the order of the American dream: red, white, blue, red, white, blue... Here the first edge could be any color and then the colors have to cycle through the American dream thereafter. We say that a vertex s can **patriotically reach** a vertex t if there is an American walk from s to t .

Consider the problem of deciding whether a vertex s can patriotically reach t . For this problem, either (a) design and analyze a polynomial time algorithm, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.⁴

Exercise 6.5. Let $G = (V, E)$ be a patriotic directed graph where all the edges are colored either red, white, or blue. An **un-American walk** is a walk that never cycles through the American dream; that is, a walk where no three consecutive edges in the walk have colors that form any of the following three sequences: (red, white, blue), (white, blue, red), or (blue, red, white). We say that a vertex s can **un-patriotically reach** a vertex t if we can walk from s to t via an un-American walk.

Consider the problem of deciding whether a vertex s can un-patriotically reach t . For this problem, either (a) design and analyze a polynomial time algorithm, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 6.6. 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

⁴Hint: don't change the algorithm; change the graph!

connected by adding the minimum number of edges. For this problem, either (a) design and analyze a polynomial time algorithm, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 6.7. 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. For this problem, either (a) design and analyze a polynomial time algorithm, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.
2. Consider the problem of deciding if G is *strictly* half-connected. For this problem, either (a) design and analyze a polynomial time algorithm, or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 6.8. Let $G = (V, E)$ be a directed graph with m edges and n vertices, such that there is at least one edge between every pair of vertices. 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. Partial credit will be awarded to an algorithm that works in linear time on DAGS.