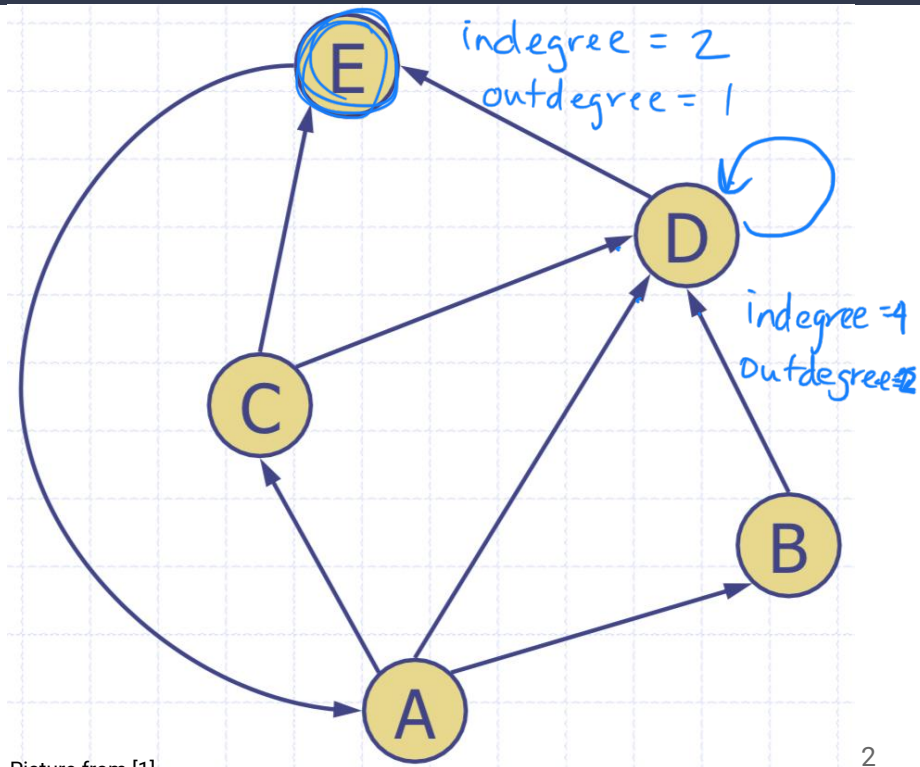
Directed Graphs

- Digraph Overview
- Directed DFS
- Strong Connectivity
- Transitive Closure
- Floyd-Warshall Algorithm
- Topological Sorting

Directed Graph (Digraph)

- A **digraph** is a graph whose edges are all directed
- The **indegree** of a vertex is the number of edges pointing to the vertex
- The **outdegree** of a vertex is the number of edges pointing away from the vertex.

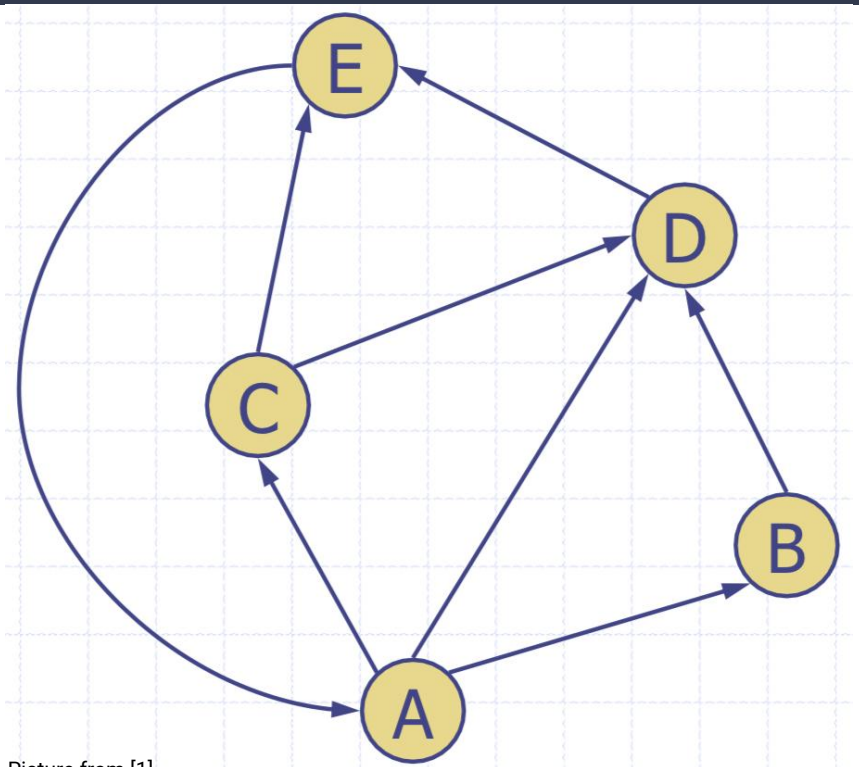
$$\sum_{v \in V} \text{indegrees} = \sum_{v \in V} \text{outdegrees} = E$$



Picture from [1]

Directed Graph (Digraph)

- A **directed path** is a sequence of vertices in a digraph such that there is a directed edge from each vertex to the next vertex in the sequence (e.g. A-B-D-E)
- A **directed cycle** is a directed path whose first and last vertices are the same (e.g. (A-D-E-A)
- **simple**: no repeated vertices or edges
- **length**: number of edges

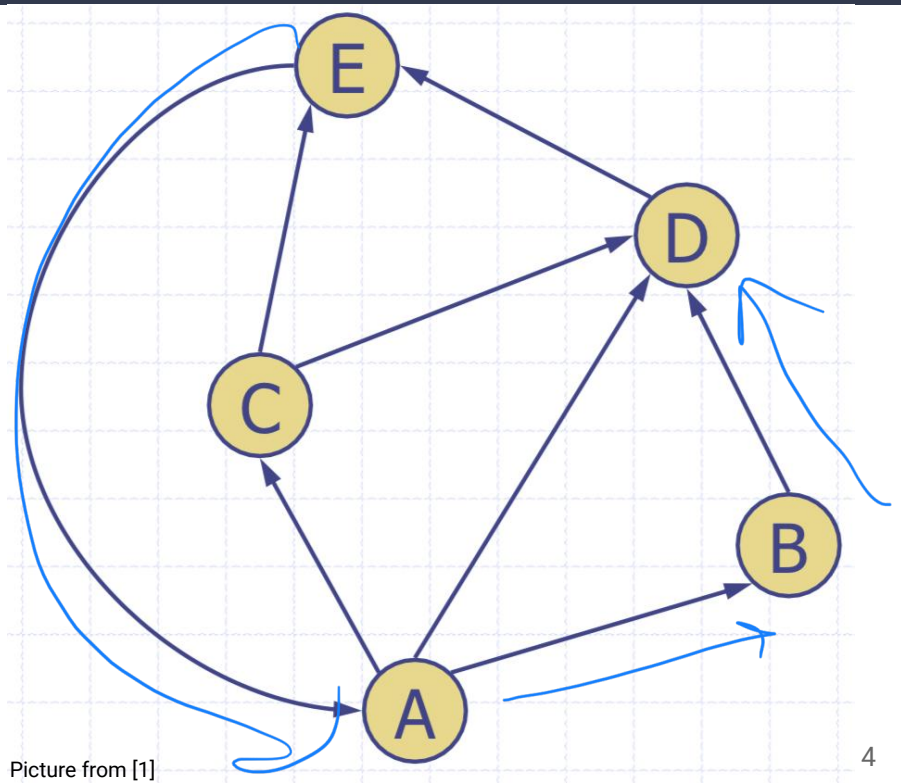


Picture from [1]

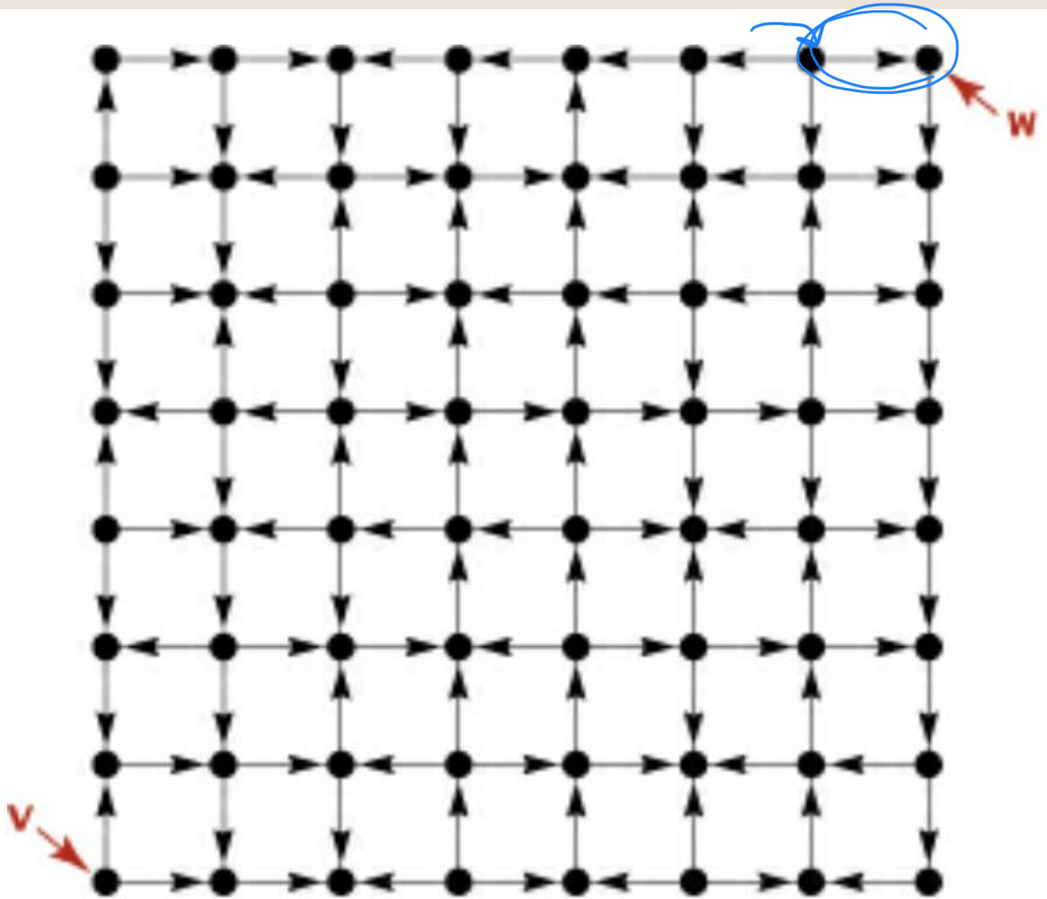
Directed Graph (Digraph)

- The edge (*u,v*) means that the edge is going out of *u* and into *v*
- A vertex *u* is **reachable** from vertex *v* iff there is a directed path from *v* to *u*

EXAMPLE: Is **D** reachable from **E**?



Picture from [1]



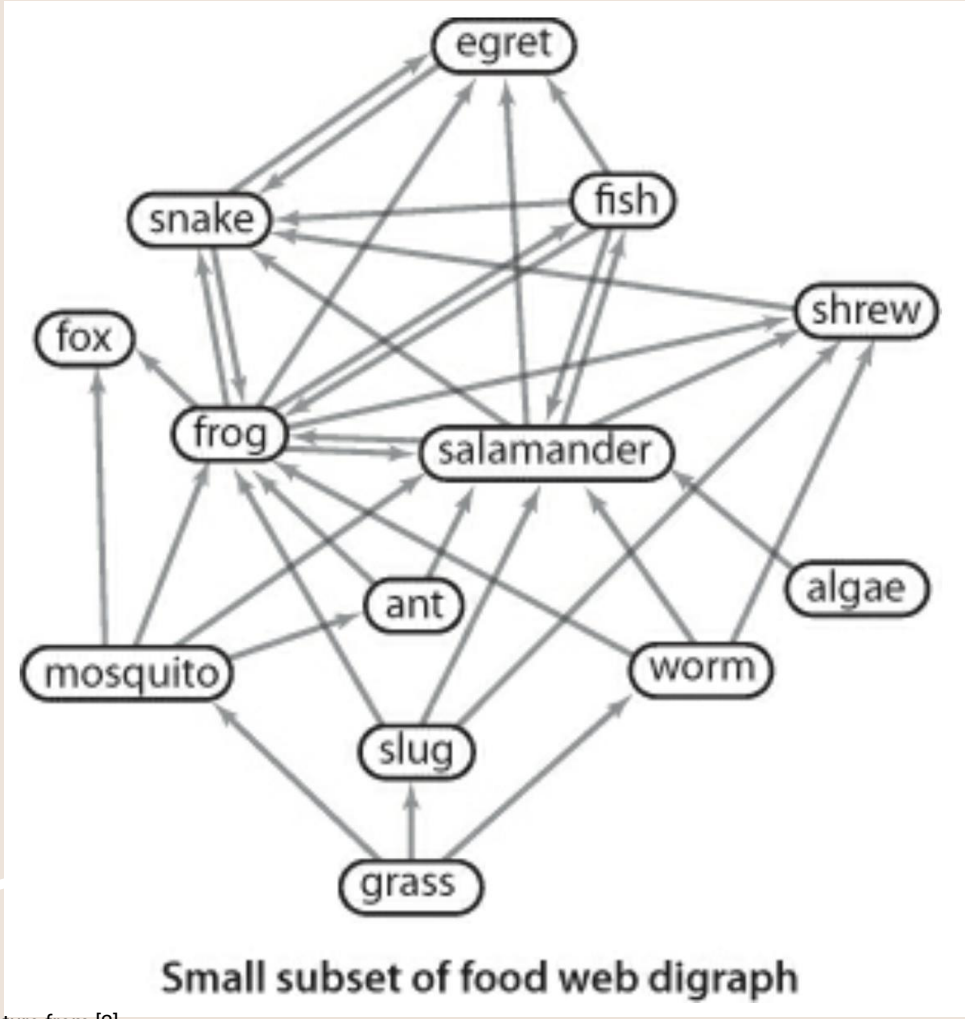
Is w reachable from v in this digraph?

Picture from [2]

application	vertex	edge
food web	species	predator-prey
internet content	page	hyperlink
program	module	external reference
cellphone	phone	call
scholarship	paper	citation
financial	stock	transaction
internet	machine	connection

Typical digraph applications

Picture from [2]



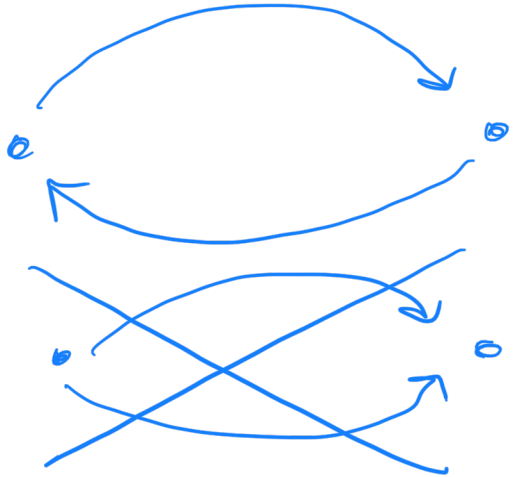
Small subset of food web digraph

Picture from [2]

- An edge (u, v) means "u is eaten by v"
- Edges are directed so (u,v) and (v,u) are not the same edge
- How can we represent digraphs?
  - Edge list
  - Adjacency matrix
  - Adjacency lists
- What would be different about this adjacency matrix and one for an undirected graph?
- What would be the exact space requirement for an adjacency list in terms of |V| and |E|?

Simple Digraph

- A **simple** digraph has no parallel edges and no self-loops
- In a simple digraph, **E <= V(V-1)**
- How is that different from a simple undirected graph and why?



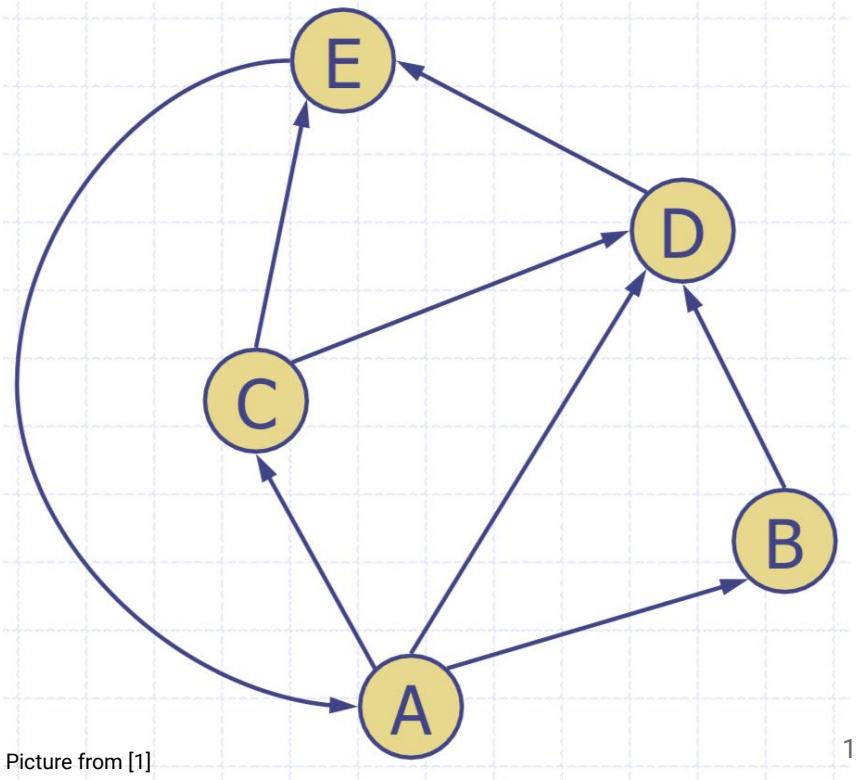
public class	Digraph	
	Digraph(int V)	*create a V-vertex digraph with no edges*
	Digraph(In in)	*read a digraph from input stream* in
int	V()	*number of vertices*
int	E()	*number of edges*
void	addEdge(int v, int w)	*add edge* v->w *to this digraph*
Iterable<Integer>	adj(int v)	*vertices connected to* v *by edges pointing from* v
Digraph	reverse()	*reverse of this digraph*
String	toString()	*string representation*

API for a digraph

Picture from [2]

Digraph API and Representation

What is a (possible) disadvantage of using a regular adjacency list representation?

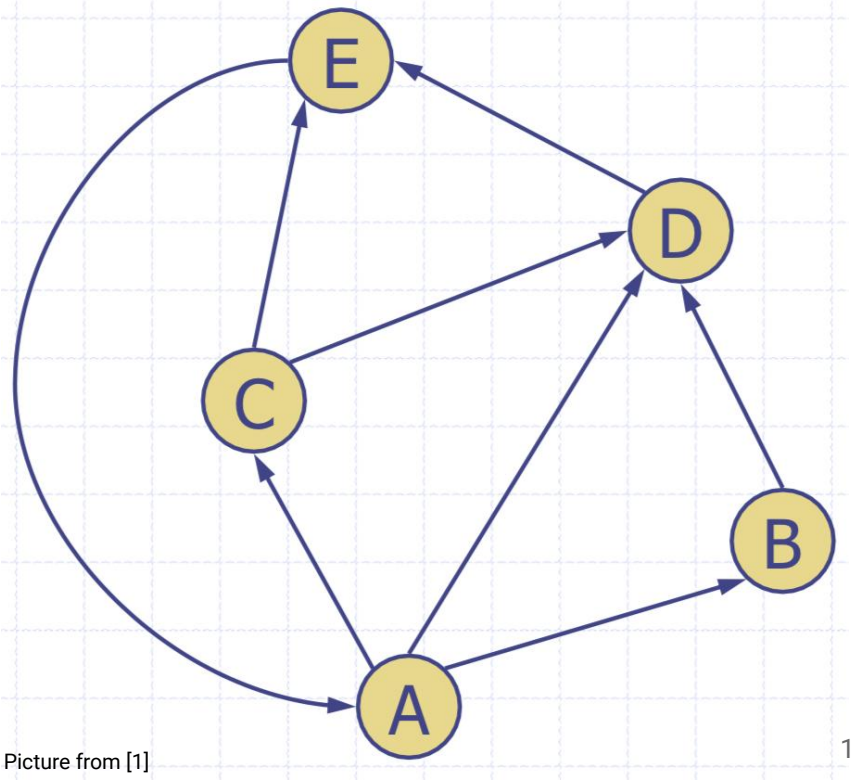


Picture from [1]

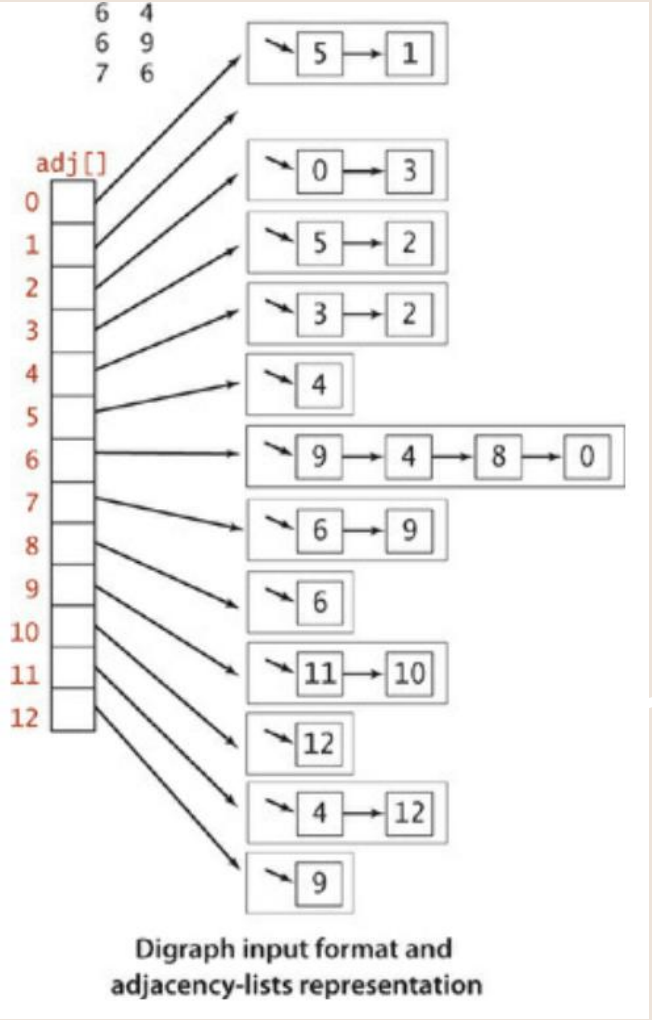
Digraph API and Representation

What is a (possible) disadvantage of using a regular adjacency list representation?

With a regular adjacency list representation, it is easy to get the **outdegree** of a graph but not as easy to get the **indegree** of a graph.



Picture from [1]

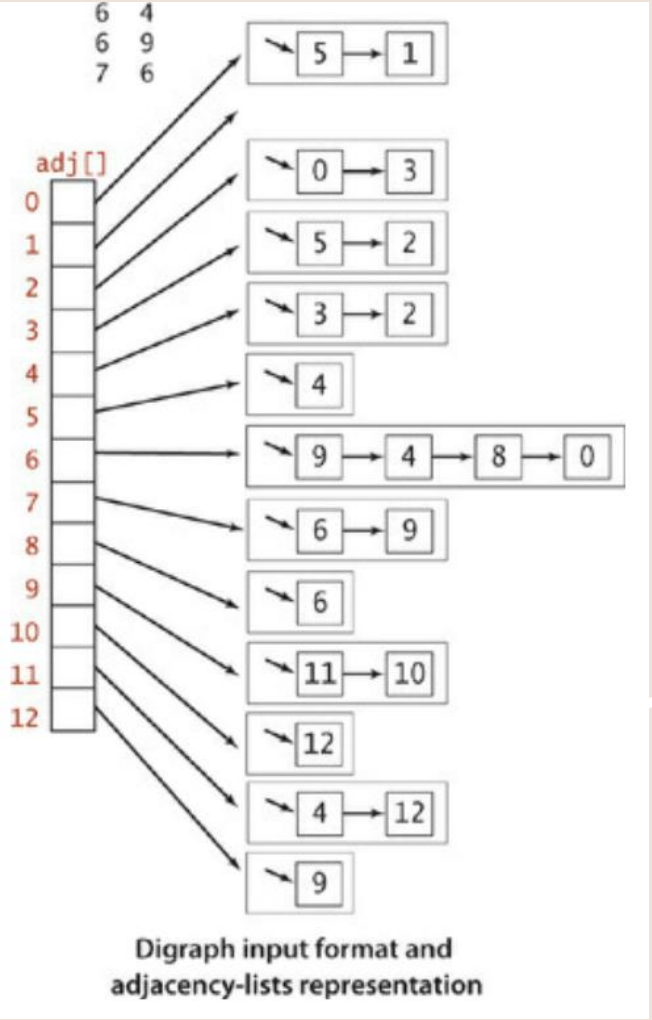


Digraph input format and adjacency-lists representation

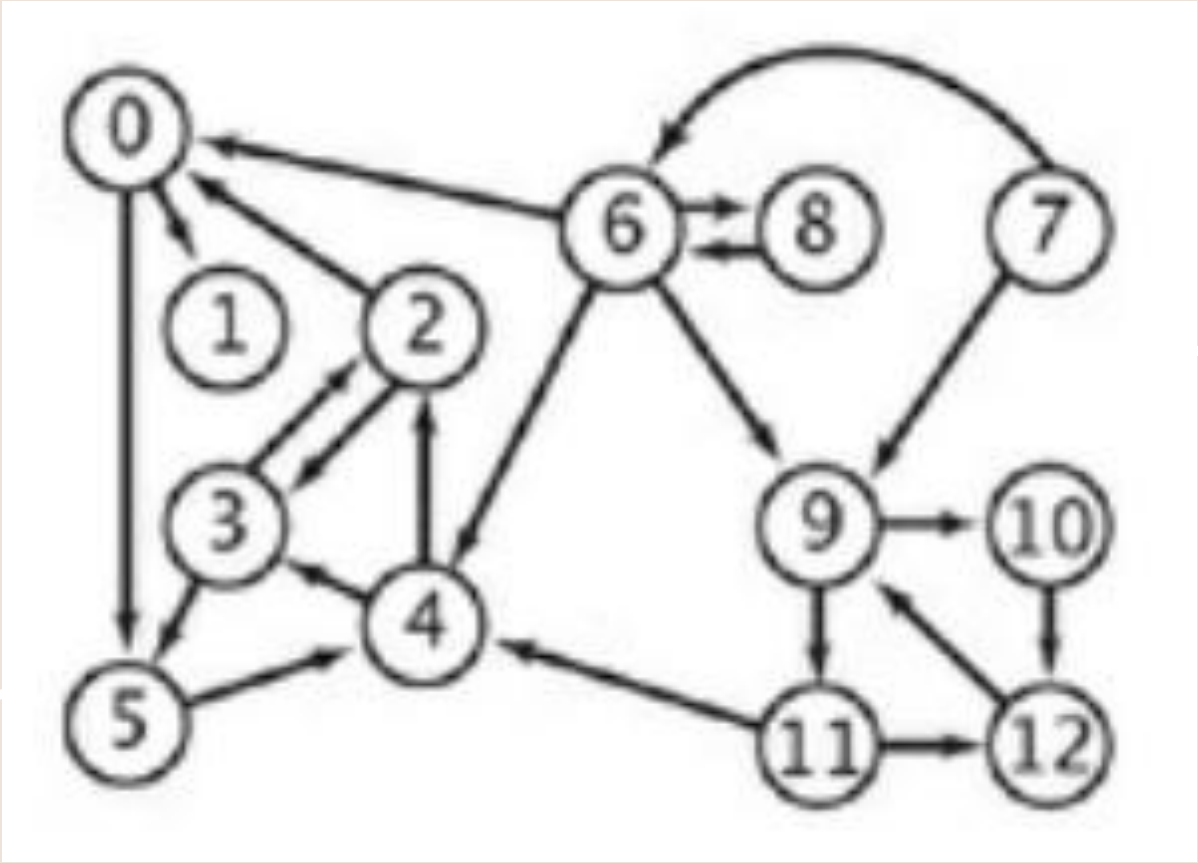
With a regular adjacency list representation, it is easy to get the **outdegree** of a graph but not as easy to get the **indegree** of a graph.

- We could maintain two adjacency lists per vertex, one for the edges going out and one for the edges going in (What would be the space?)
- We could calculate the **indegree** of a vertex by analyzing the **reverse** of the graph

Picture from [2]



Digraph input format and adjacency-lists representation

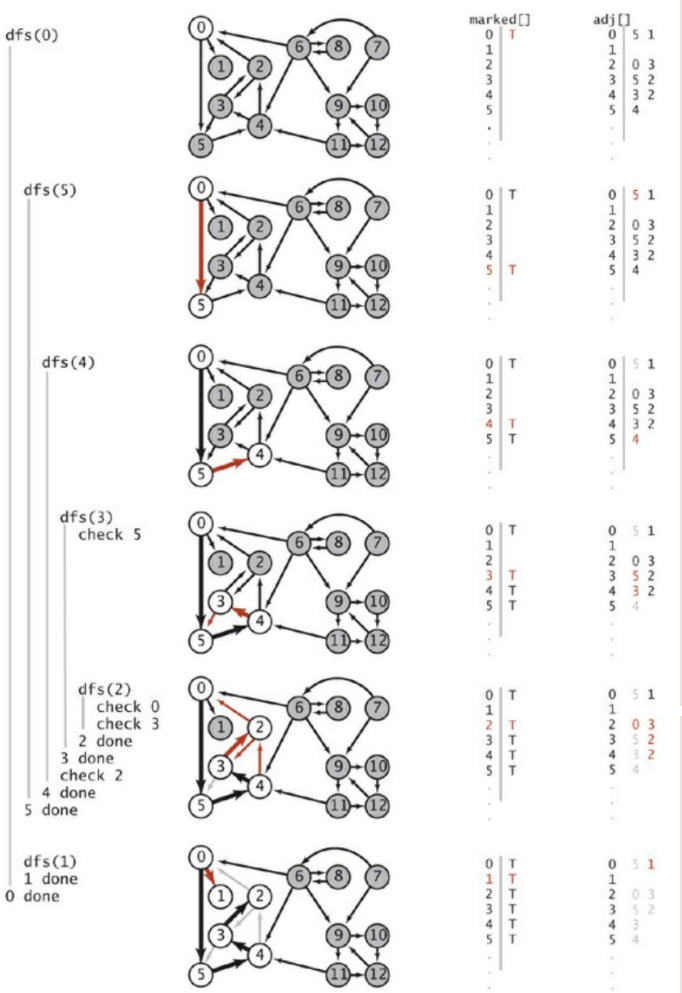


Pictures from [2]

Single-source Reachability

Given a digraph and a source vertex *s*, determine if there is a directed path from *s* to a target vertex *t*.

But how???



Trace of depth-first search to find vertices reachable from vertex 0 in a digraph

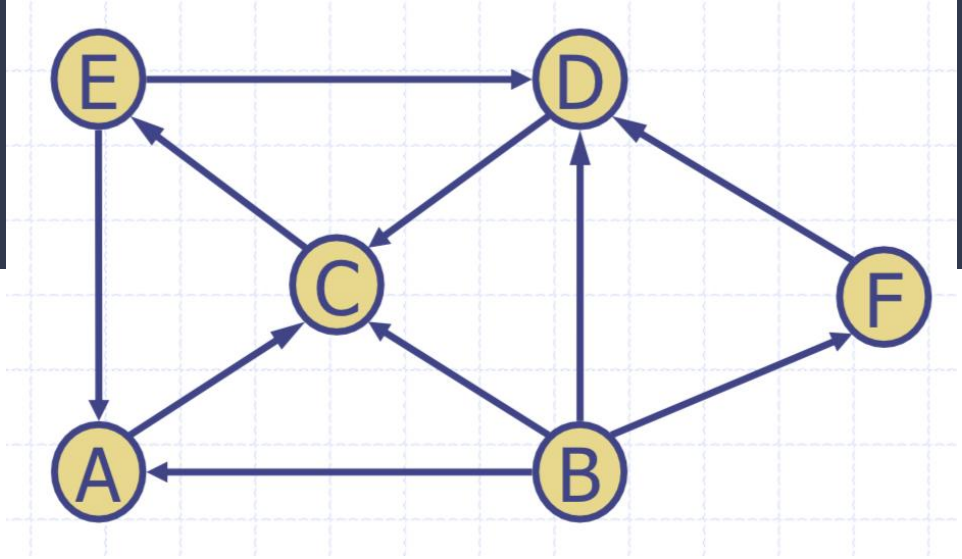
Directed DFS

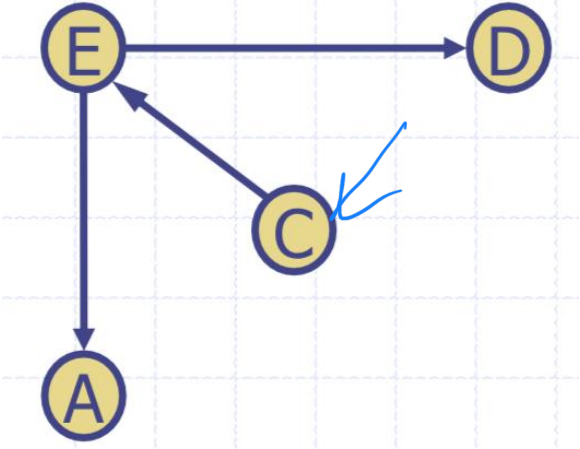
Note that with a directed graph, we would benefit from a new version of the notion of "connectedness" because *connected* no longer implies *reachability*.

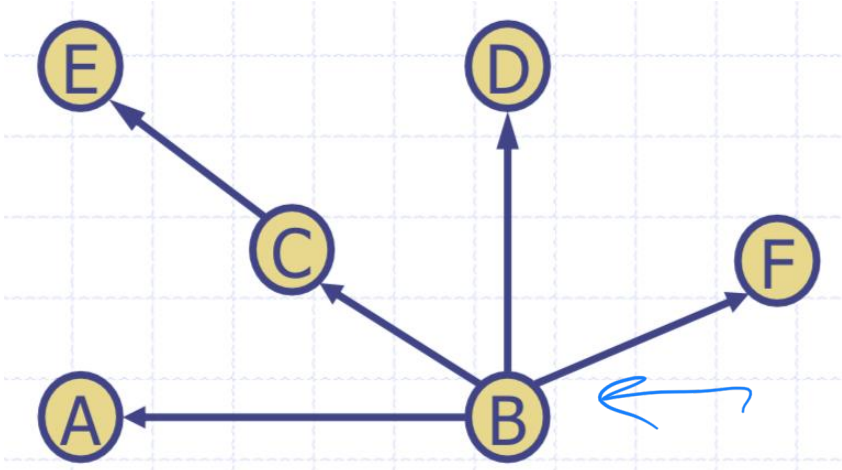
Pictures from [2]

Directed DFS

Given a source *v*, we can discover all the vertices that are reachable from *v*.

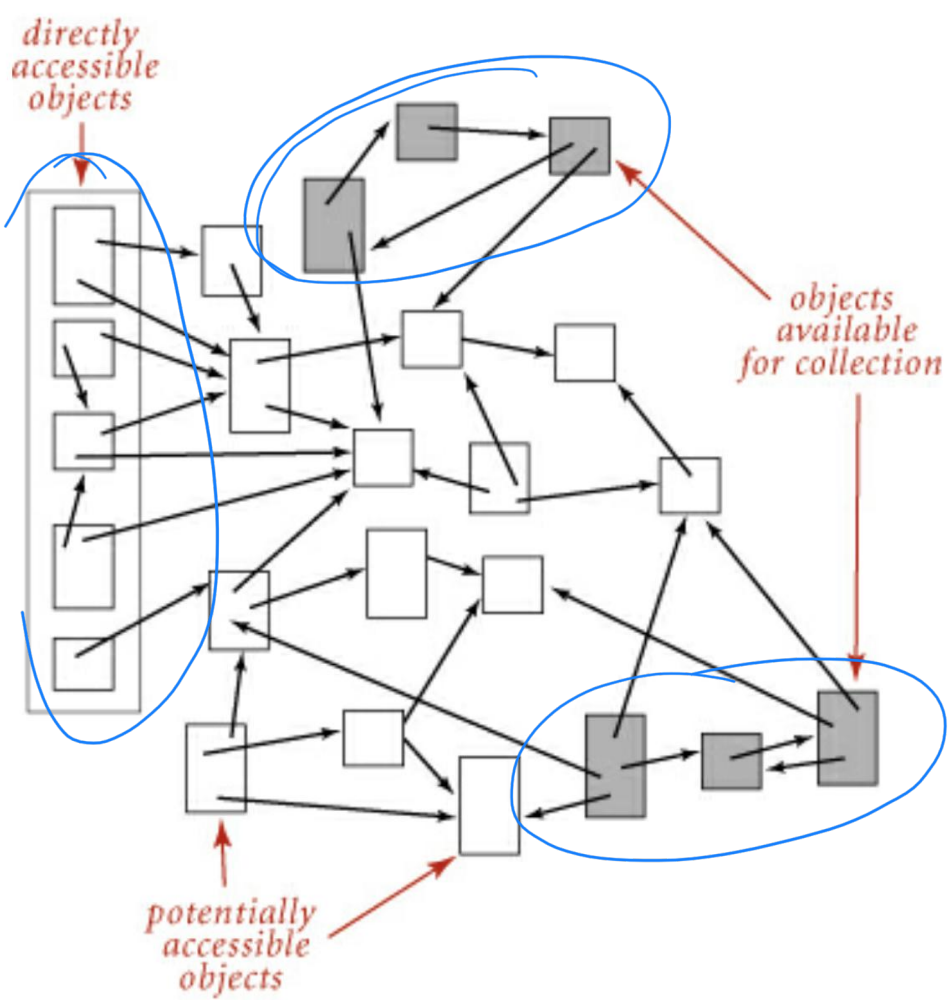






Pictures from [1]

An application: garbage collection



Garbage collection scenario

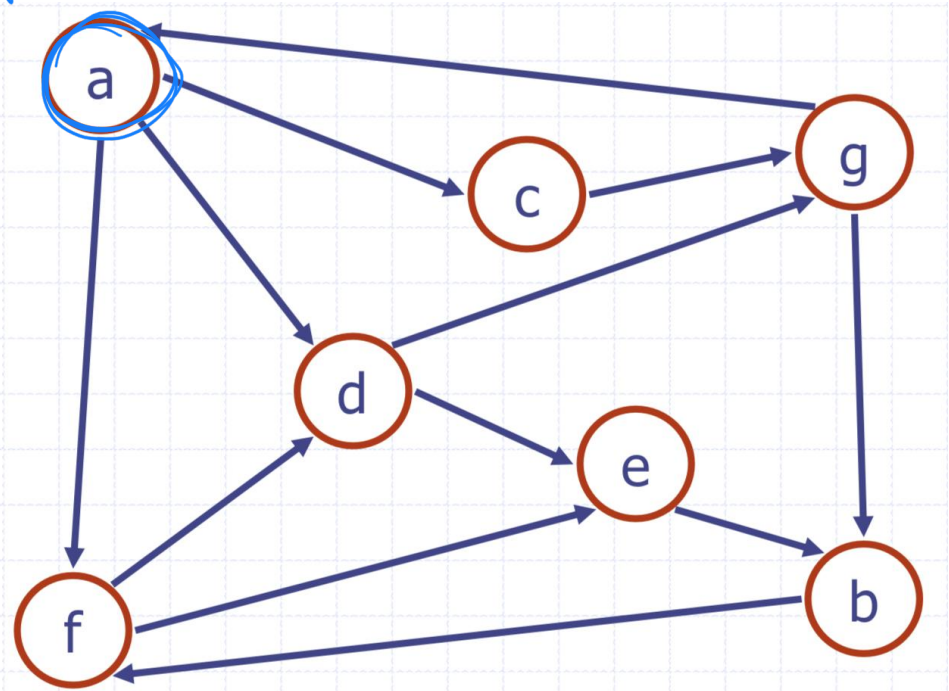
Picture from [2]

Strong Connectivitiy

DFS

A graph is **strongly connected** if every vertex is reachable from every other vertex (similar to the idea of connectivity in an undirected graph.)

How can we determine if a graph is strongly connected or not?



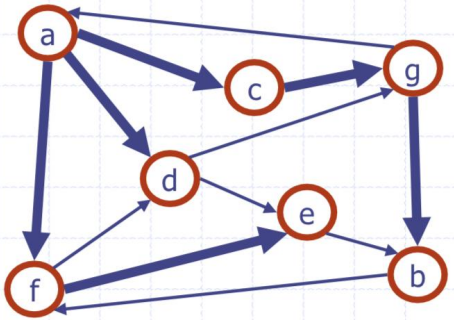
Picture from [1]

Strong Connectivity Algorithm

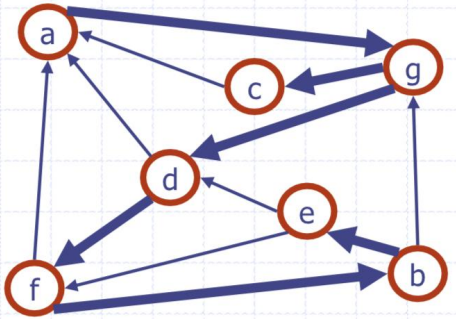
- Pick a vertex v in G.
- Perform a DFS from v in G.
  - If there's a w not visited, print "no".
- Let G' be G with edges reversed.
- Perform a DFS from v in G'.
  - If there's a w not visited, print "no".
  - Else, print "yes".

- Running time: O(n+m).

G:



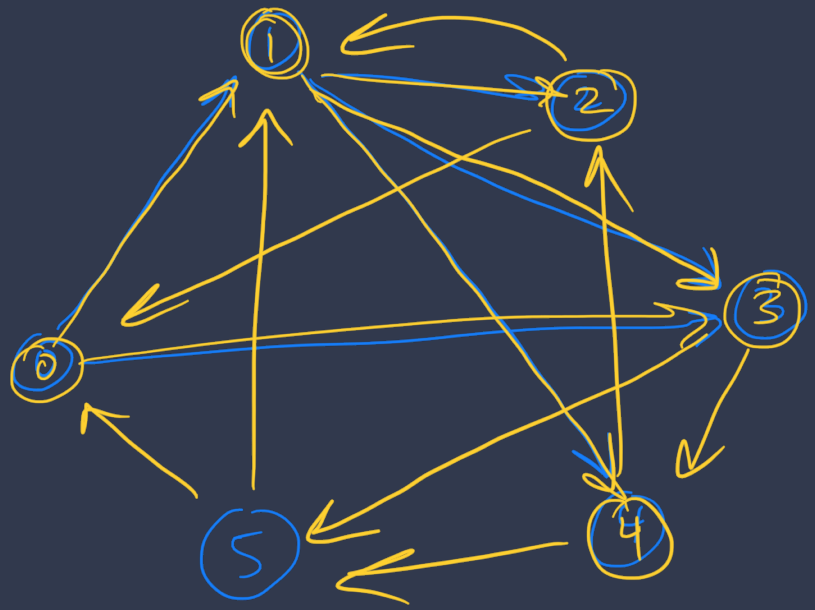
G':

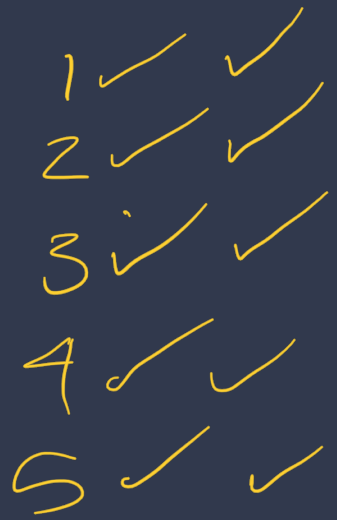


Picture from [1]

Example. Is this graph strongly connected?

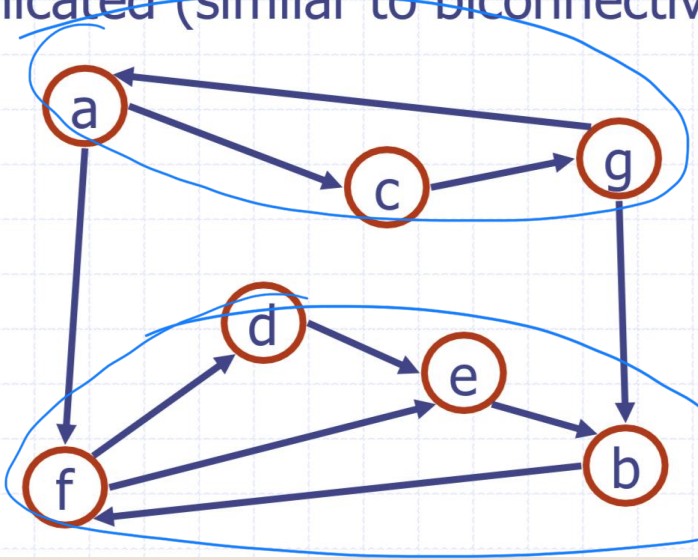
0	1, 3
1	2, 3, 4
2	0, 1
3	4, 5
4	2, 5
5	0, 1





Strongly Connected Components

- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in O(n+m) time using DFS, but is more complicated (similar to biconnectivity).



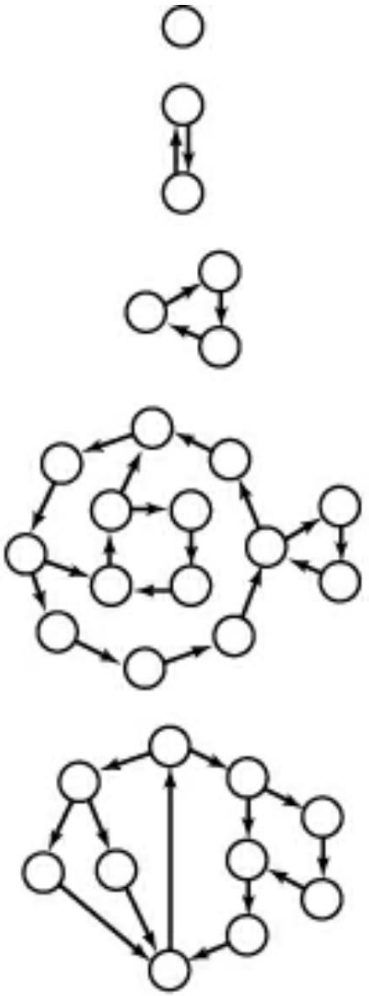
{ a , c , g }

{ f , d , e , b }

Picture from [1]

Properties of Strong Connectivity

- **Reflexive**: Every vertex is strongly connected to itself
- **Symmetric**: If *v* is strongly connected to *u*, then *u* is strongly connected to *v*
- **Transitive**: If *v* is strongly connected to *u* and *u* is strongly connected to *w*, then *v* is strongly connected to *w*



Strongly connected digraphs

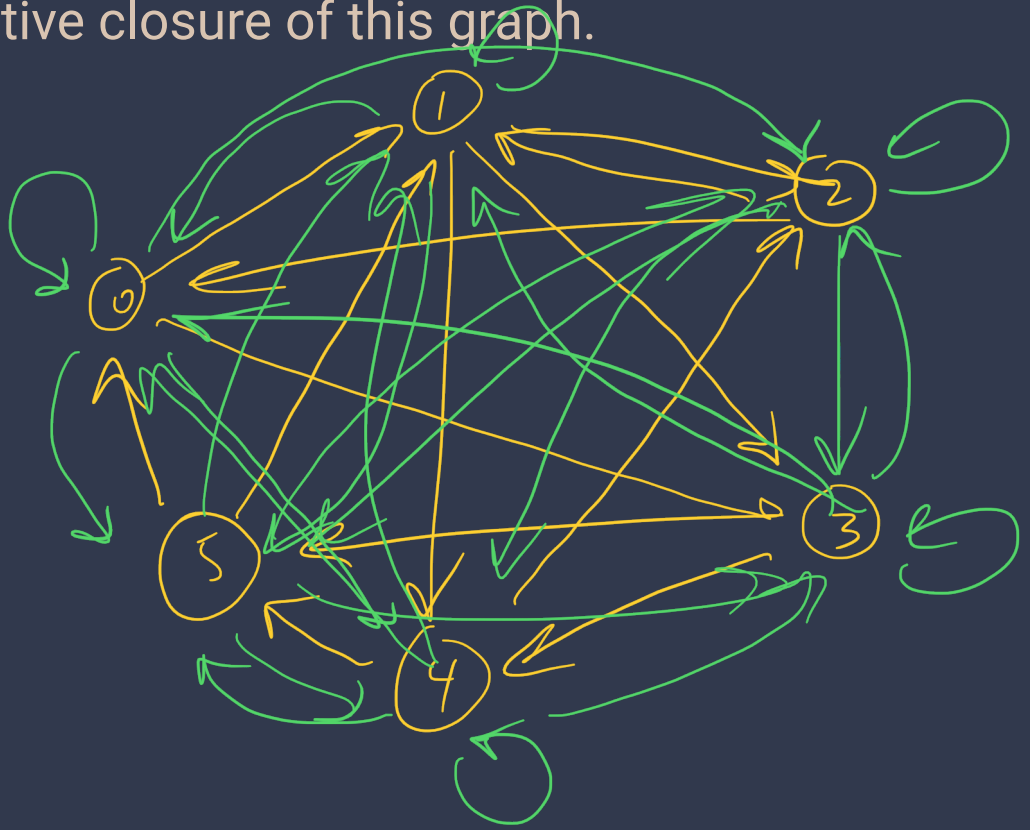
Transitive Closure

G* is the *transitive closure* of a directed graph G if:

- *G** has the same vertices as *G*
- *G** contains a directed edge *(u, v)* if and only if there is a directed path from *u* to *v* in *G*

Example. Draw the transitive closure of this graph.

0	1, 3
1	2, 3, 4
2	0, 1
3	4, 5
4	2, 5
5	0, 1



Transitive Closure and self-loops...

- Variation 1 (Sedgewick & Wayne): Automatically add 1's down the diagonal because every vertex is reachable from itself (i.e. every vertex has a self-loop in the transitive closure.)
- Variation 2 (Goodrich & Tamassia): Set the diagonal to all 0's unless a self-loop exists in the original graph.
- Variation 3 (Warshall): The transitive closure will have 1's in the diagonal (i.e. self-loops) if:
  - there is a self-loop in the original graph
  - there is a cycle in the original graph containing that vertex

In this class, we use Variation 3.

How can we compute the transitive closure of a graph?

How can we compute the transitive closure of a graph?

Run DFS on all vertices:

$O(n(n+m)) \rightarrow O(n^3)$ for dense graphs...

$n^2 + nm$

Floyd-Warshall Transitive Closure

Algorithm FloydWarshall(M)

Input: **M**, the adjacency matrix of a graph **G**

Output: The transitive closure **G*** of the graph **G**

$R^{(0)} \leftarrow M$

for $k \leftarrow 1$ **to** n **do**

  for $i \leftarrow 1$ **to** n **do**

    for $j \leftarrow 1$ **to** n **do**

      $R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j]$ **or** $(R^{(k-1)}[i,k]$ **and** $R^{(k-1)}[k,j])$

return $R^{(n)}$

Example. Compute the transitive closure of the graph below using the Floyd-Warshall Algorithm.

k

	0	1	2
0	0	1	1
1	0	0	0
2	1	1	0

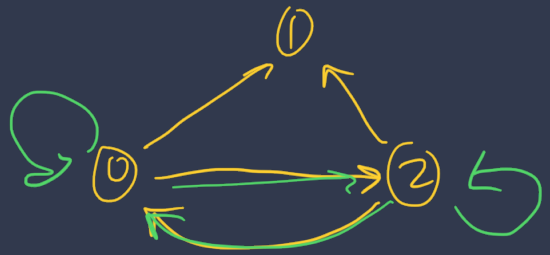
k

0	1	1
0	0	0
1	1	1

k

0	1	1
0	0	0
1	1	1

1	1	1
0	0	0
1	1	1



Floyd-Warshall Algorithm Discussion

- Assuming an adjacency matrix representation, what is the space requirement for the algorithm?

- What is the runtime of the algorithm?

- Assuming no self-loops in the original graph, what does a self-loop (a 1 in the diagonal of the matrix) in the transitive closure graph tell us?

- What does it mean if the resulting matrix is all 1's?

Floyd-Warshall Algorithm Discussion

- Assuming an adjacency matrix representation, what is the space requirement for the algorithm?
  $O(V^2)$

- What is the runtime of the algorithm?
  $O(V^3)$

- Assuming no self-loops in the original graph, what does a self-loop (a 1 in the diagonal of the matrix) in the transitive closure graph tell us?
  There is a cycle in the original graph starting and ending at that vertex.

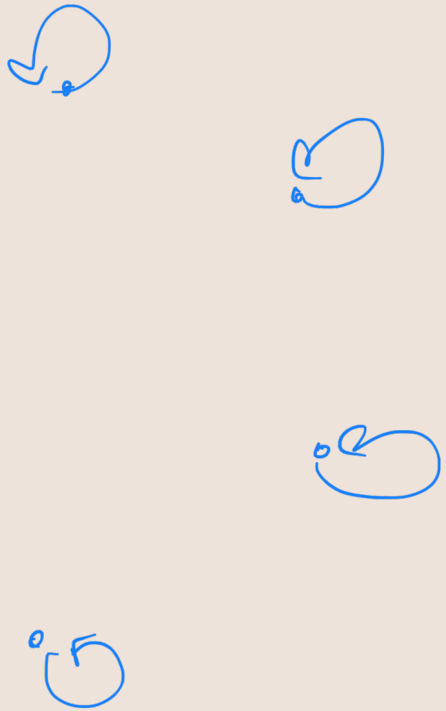
- What does it mean if the resulting matrix is all 1's?
  The original graph is strongly connected.

Is this a possible transitive closure of a directed graph?

0	1	1	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Is this a possible transitive closure of a directed graph?

1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1



Is this a possible transitive closure of a directed graph?

1	1	1	1	1
0	1	1	1	1
0	0	1	1	1
0	0	0	1	1
0	0	0	0	1

Is this a possible transitive closure of a directed graph?

1	0	1	0	1
0	0	1	0	1
1	1	1	0	1
0	0	0	0	1
1	1	1	1	1

DAGs and Topological Ordering

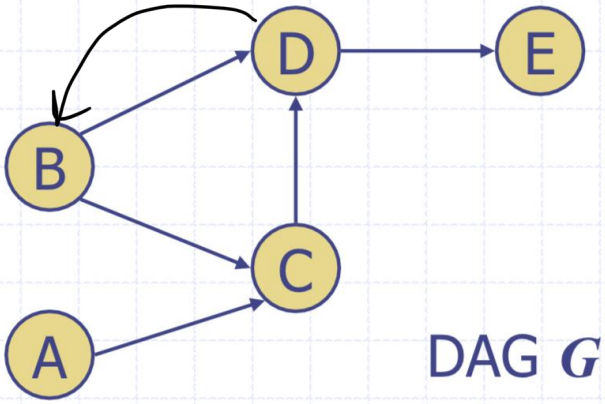
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

  $v_1, \ldots, v_n$

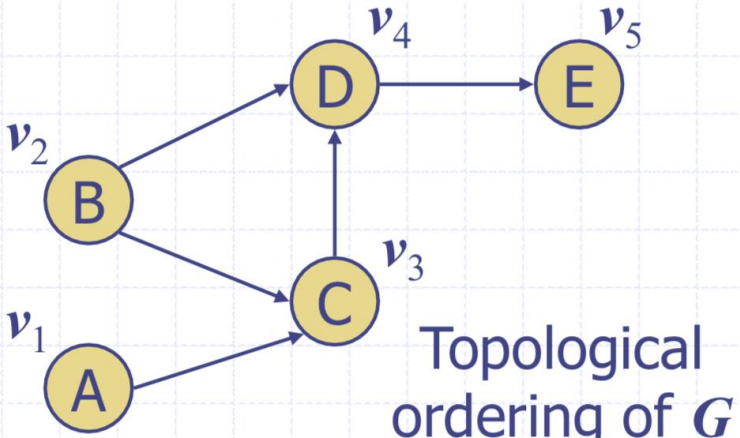
  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering of a task sequence satisfies the precedence constraints

Theorem

A digraph admits a topological ordering if and only if it is a DAG

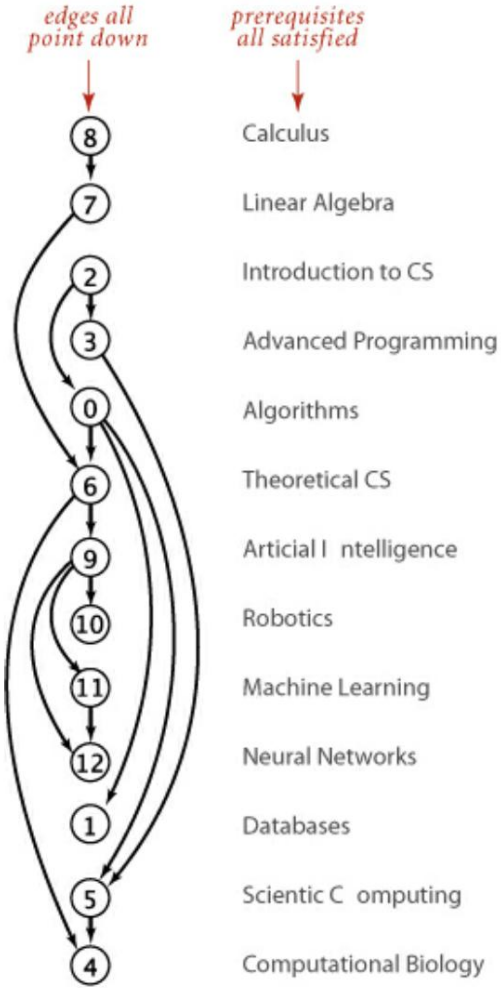


DAG G



Topological ordering of G

Picture from [1]



Topological sort

Picture from [2]

Topological Ordering is especially useful in scheduling problems...

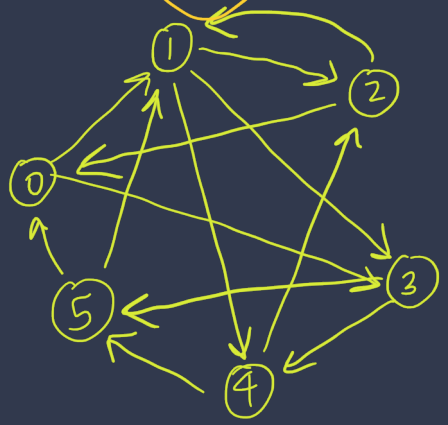
Topological Sorting

```
Method TopologicalSort(G)
    H ← G          // Temporary copy of G
    n ← G.numVertices()
    while H is not empty do
        Let v be a vertex with no outgoing edges
        Label v ← n
        n ← n - 1
        Remove v from H
```

Picture from [1]

Example. Find all the topological orders of the graph below.

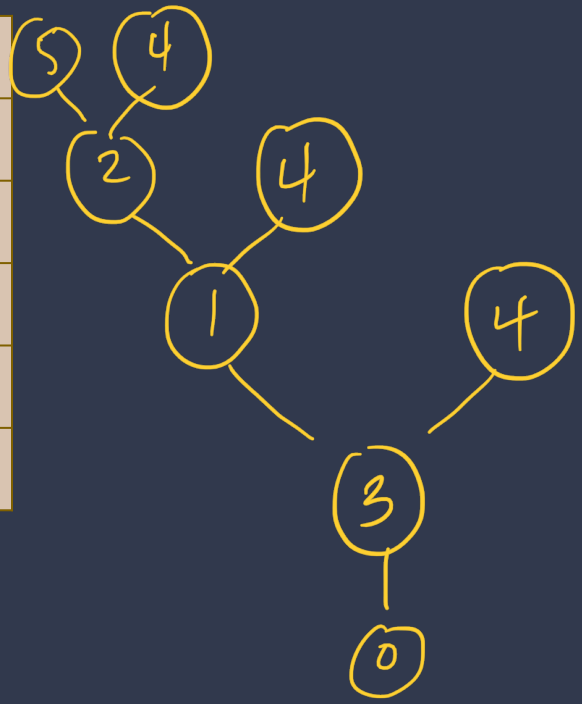
0	1, 3
1	2, 3, 4
2	0, 1
3	4, 5
4	2, 5
5	0, 1



NONE

Example. Find all the topological orders of the graph below.

0	
1	3
2	1, 3
3	
4	0, 3
5	2



3

Topological Sorting

```
Method TopologicalSort(G)
    H ← G          // Temporary copy of G
    n ← G.numVertices()
    while H is not empty do
        Let v be a vertex with no outgoing edges
        Label v ← n
        n ← n - 1
        Remove v from H
```

Picture from [1]

Runtime: **O(V + E)**

How do we actually implement it given an adjacency list representation?

Topological Sorting with DFS

- Run DFS using a Stack to keep track of the current path (this will also determine if there are any cycles.)
- Return vertices in the opposite order of the vertices that are “done” first--done meaning the recursive call is completed (i.e. there are no outgoing edges to consider.)
- May need to run DFS again on any unvisited vertices, but this is a separate run, so the runtime is added not multiplied.
- Total runtime: O(V + E)

References

[1] Tamassia and Goodrich
[2] Sedgewick and Wayne