MONASH
INFORMATION
TECHNOLOGY

FIT2004
Algorithms and Data Structures

Ian Wern Han Lim
lim.wern.han@monash.edu

Referencing materials by
Nathan Companez, Aamir Cheema, Arun Konagurthu and Lloyd Allison

Faculty of Information Technology, Monash University

Ready?

Agenda

- Directed Acyclic Graph (DAG)

Agenda

- Directed Acyclic Graph (DAG)
- Topological Sort
  - Kahn’s algorithm
  - Depth-First Search (DFS) modification

Let us begin…

Directed Acyclic Graph (DAG)
What is it?

- Graphs are very commonly used to model real world scenario
  - One of which is a DAG

Directed Acyclic Graph (DAG)
What is it?

- Graphs are very commonly used to model real world scenario
  - One of which is a DAG

- What is a DAG?

Directed Acyclic Graph (DAG)
What is it?

MONASH University

- Graphs are very commonly used to model real world scenario
  - One of which is a DAG
- What is a DAG?
  - Directed

Directed Acyclic Graph (DAG)
What is it?

- Graphs are very commonly used to model real world scenario
  - One of which is a DAG
- What is a DAG?
  - Directed
  - Acyclic

Directed Acyclic Graph (DAG)

What is it?

- Graphs are very commonly used to model real world scenario
  - One of which is a DAG

- What is a DAG?
  - Directed directed edeg
  - Acyclic no cyclic
  - and of course it is a Graph…

Directed Acyclic Graph (DAG)

What is it?

- Graphs are very commonly used to model real world scenario
  - One of which is a DAG
  - Can you give me an example of a real world DAG?

- What is a DAG?
  - Directed
  - Acyclic
  - and of course it is a Graph…

Directed Acyclic Graph (DAG)
What is it?

MONASH University

- Graphs are very commonly used to model real world scenario
  - One of which is a DAG
  - Can you give me an example of a real world DAG?
  - Can you give me an example of a real world non-DAG?

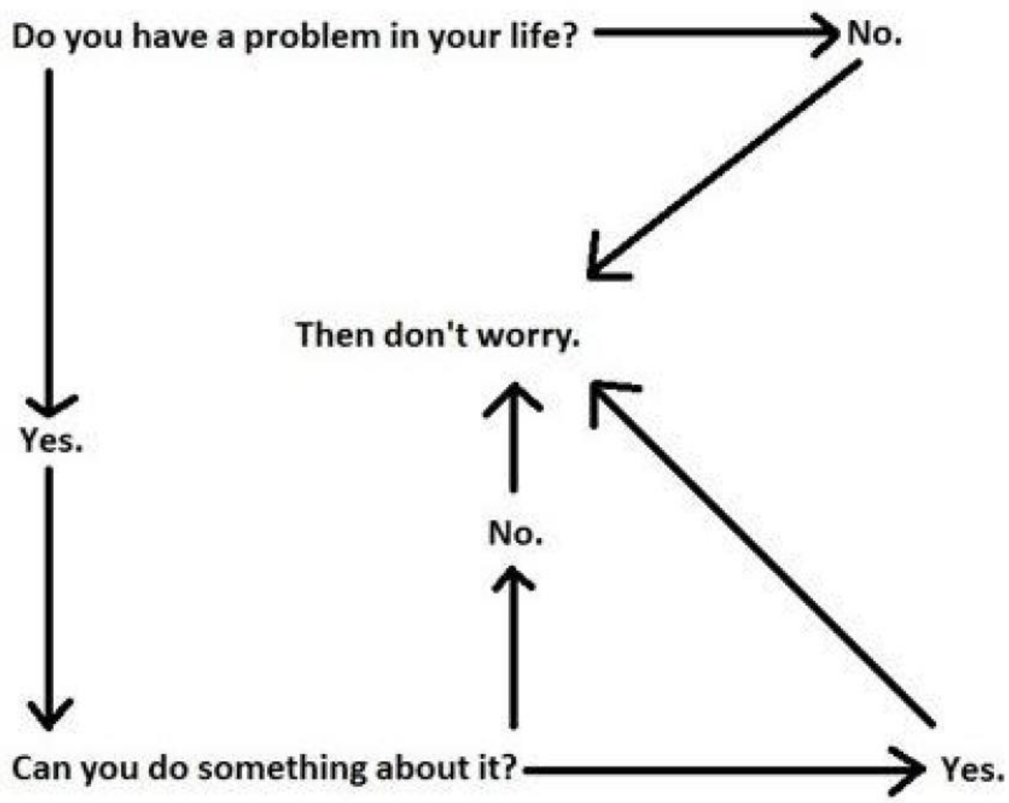
have cycle

- What is a DAG?
  - Directed
  - Acyclic
  - and of course it is a Graph…

Directed Acyclic Graph (DAG)

What is it?

- A real world DAG



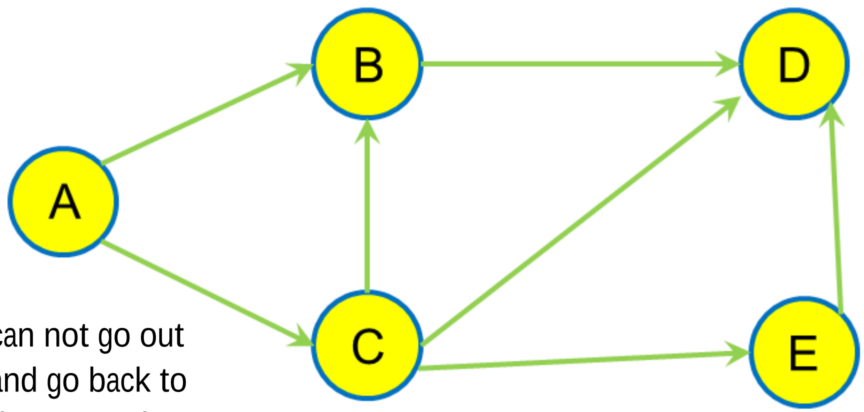
Directed Acyclic Graph (DAG)
What is it?

- A real world not DAG

Directed Acyclic Graph (DAG)
What is it?

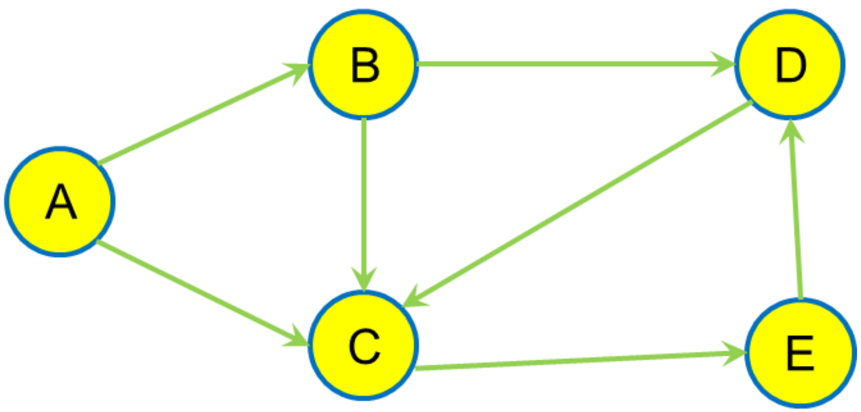
- Which graph is a DAG?
  – And where is the cycle?

DAG



can not go out
and go back to
the vertex that starts

Graph 1



Graph 2

Questions?

Directed Acyclic Graph (DAG)

What is it for?

MONASH University

- Prerequisite mapping

Directed Acyclic Graph (DAG)

What is it for?

- Prerequisite mapping
  - If I have a directed edge from A to B, this means I need A before B

Directed Acyclic Graph (DAG)
What is it for?

MONASH University

- Prerequisite mapping
  - If I have a directed edge from A to B, this means I need A before B
    - Common in project management
    - Common in talent/skill trees!

Directed Acyclic Graph (DAG)

What is it for?

- Prerequisite mapping
  - If I have a directed edge from A to B, this means I need A before B
    - Common in project management
    - Common in talent/skill trees!
    - Your university units!
      - Pass 1045
      - Pass 1008
      - Pass 2004
      - Pass 3155

Directed Acyclic Graph (DAG)

What is it for?

- So for an edge <A,B>

Directed Acyclic Graph (DAG)

What is it for?

- So for an edge <A,B>
  - A is a prerequisite for B
  - A is an ancestor of B
  - A is the subset of B
  - A is ordered before B

Directed Acyclic Graph (DAG)
What is it for?

- So for an edge <A,B>
  - A is a prerequisite for B
  - A is an ancestor of B
  - A is the subset of B
  - A is ordered before B
    - This enable us to sort a DAG

Questions?

Topological Sort

Ordering of Vertices

- A topological sort
  - 排列
    Permutation of vertices in a DAG
  - Vertex U will appear before vertex V if we have edge <U,V>

order the one unit in front that is pre-requisite of the most of the units

Topological Sort

Ordering of Vertices

- A topological sort
  - Permutation of vertices in a DAG
  - Vertex U will appear before vertex V if we have edge <U,V>
  - Vertex U will appear before vertex W is we have edge <U,W>

Topological Sort

Ordering of Vertices

- A topological sort
  - Permutation of vertices in a DAG
  - Vertex U will appear before vertex V if we have edge <U,V>
  - Vertex U will appear before vertex W is we have edge <U,W>
  - But if we don’t have edge <V,W> then V and W are of the same order

Topological Sort
Ordering of Vertices

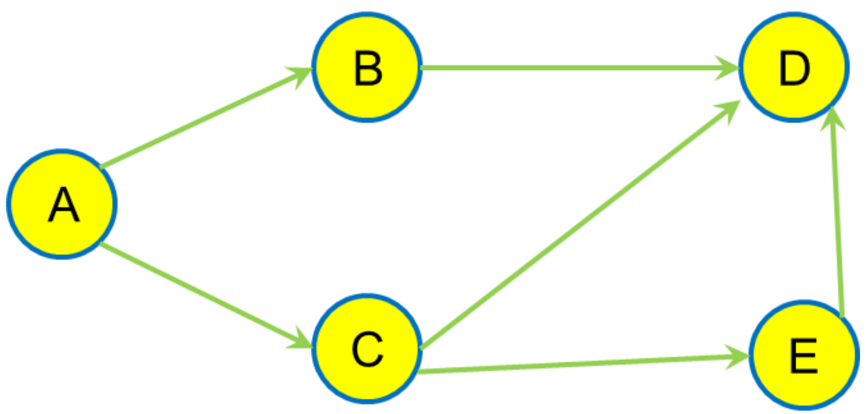
- A topological sort
  - Permutation of vertices in a DAG
  - Vertex U will appear before vertex V if we have edge <U,V>
    - U<V
  - Vertex U will appear before vertex W is we have edge <U,W>
    - U<W
  - But if we don’t have edge <V,W> then V and W are of the same order
    - V==W   U V W or U W V

Topological Sort
Ordering of Vertices

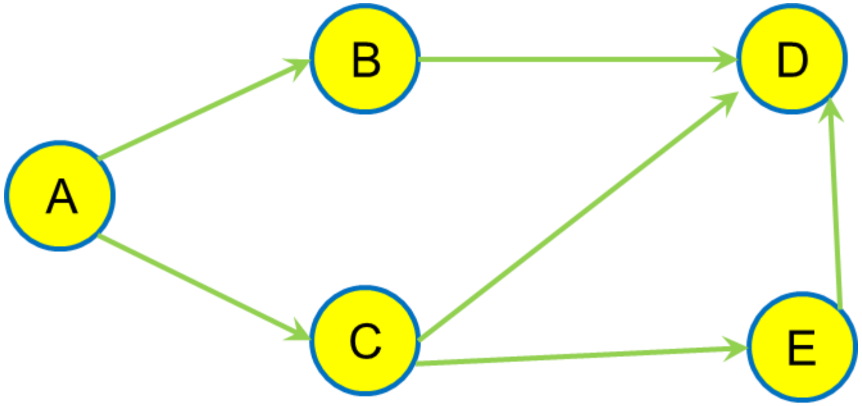
- A topological sort
  - Permutation of vertices in a DAG
  - Vertex U will appear before vertex V if we have edge <U,V>
    - U<V
  - Vertex U will appear before vertex W is we have edge <U,W>
    - U<W
  - But if we don’t have edge <V,W> then V and W are of the same order
    - V==W

- So we have a DAG of your units
- Topological sort of this DAG gives you the order of units to take!

- Which one of these are not a valid topological sort of the DAG?



- Which one of these are not a valid topological sort of the DAG?
  1. A, B, C, E, D
  2. A ,C, B, E, D
  3. A, C, E, B, D   E can be before B, as long as B before D
  4. A, B, E, C, D



- Topological sort can be done via
  - Kahn’s algorithm
  - A modified DFS

Questions?

Kahn’s Algorithm

For topological sort

- What is the concept like?

Kahn's Algorithm

For topological sort

- What is the concept like?
  - Start with vertices without incoming edges

Kahn's Algorithm

For topological sort

- What is the concept like?
  - Start with vertices without incoming edges
  - Delete all outgoing edges from the vertex

Kahn’s Algorithm
For topological sort

- What is the concept like?
  - Start with vertices without incoming edges
  - Delete all outgoing edges from the vertex
  - Add in vertices without incoming edges

Kahn's Algorithm
For topological sort

- What is the concept like?
  - Start with vertices without incoming edges
  - Delete all outgoing edges from the vertex
  - Add in vertices without incoming edges
  - Repeat!

Kahn's Algorithm

For topological sort

- Algorithm as follow

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

remove outgoing edge
only add vertex that has no incoming edge

Kahn's Algorithm

For topological sort

- Algorithm as follow
  - Let us try it out

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

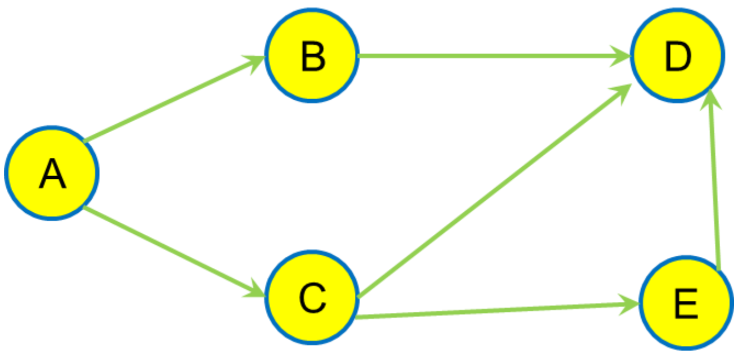
Kahn's Algorithm
For topological sort

- Algorithm as follow
  - Let us try it out

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```



sorted					
process					

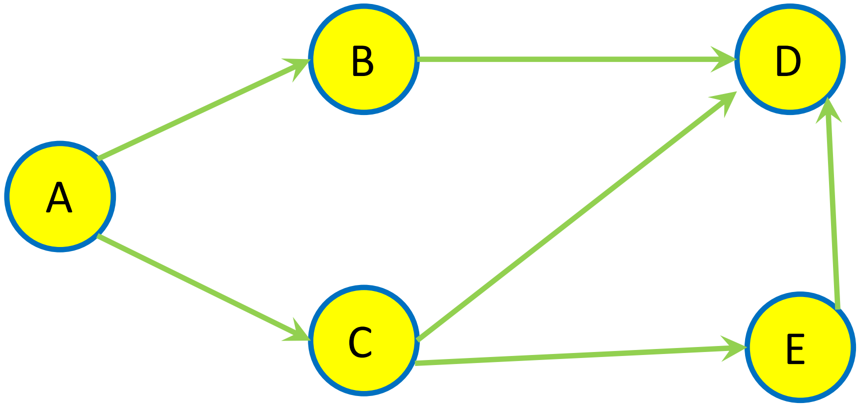
Kahn's Algorithm
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



sorted					
process	A				

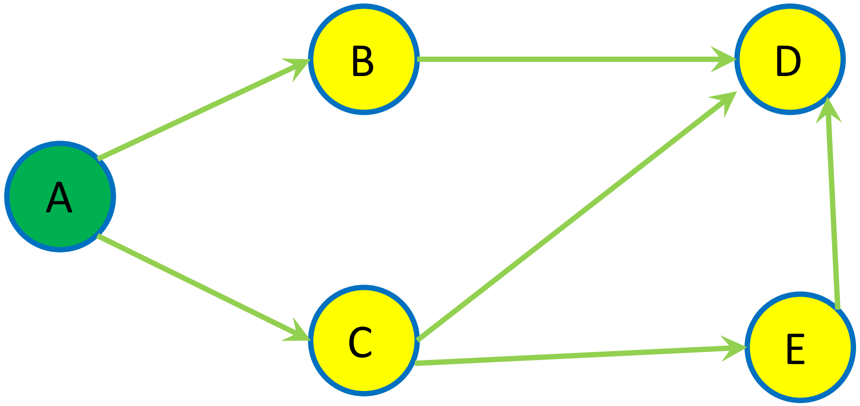
Kahn’s Algorithm
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



sorted	A				
process					

Kahn's Algorithm

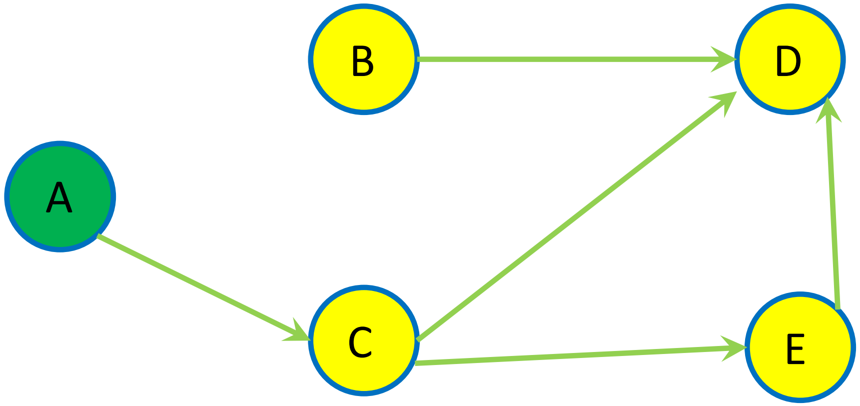
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



sorted	**A**				
process					

Kahn's Algorithm

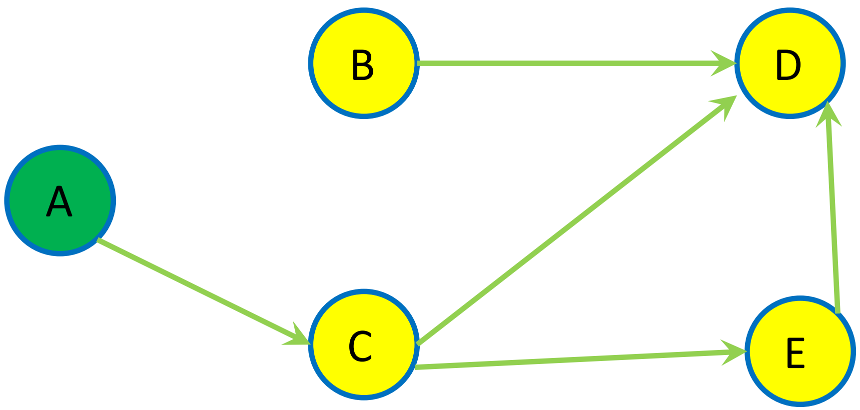
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



sorted	**A**				
process	B				

Kahn's Algorithm

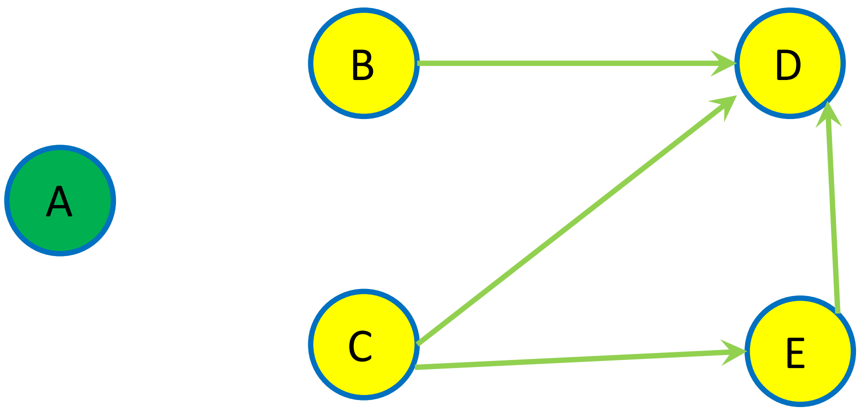
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



sorted	A				
process	B	C			

if queue, B serve first
if stact, C serve first

Kahn’s Algorithm

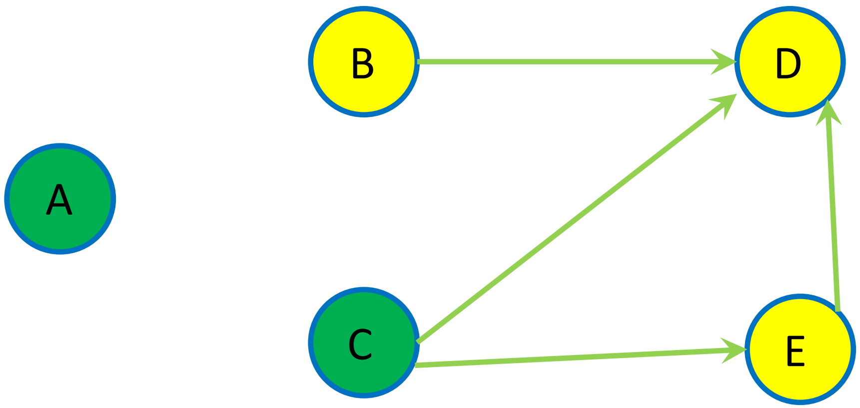
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



delete outgoing edge from sorted element: C

sorted	**A**	**C**			
process	B				

Kahn’s Algorithm

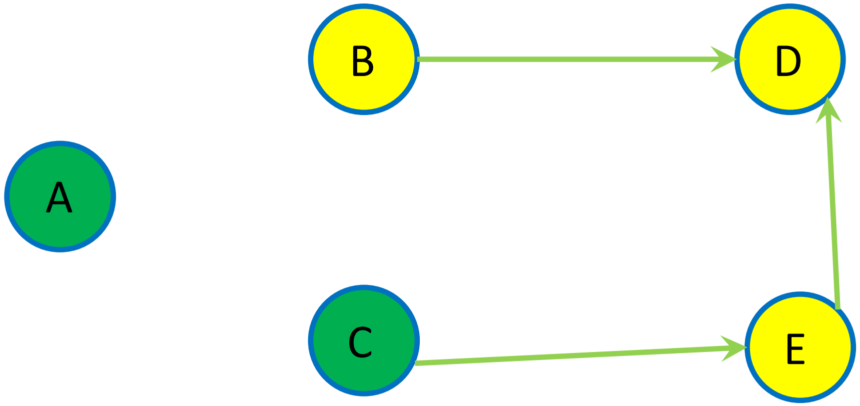
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



sorted	A	C			
process	B				

Kahn’s Algorithm

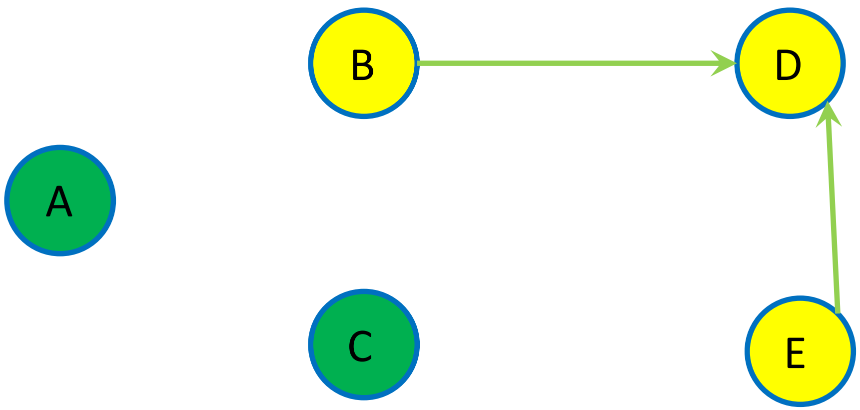
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



sorted	A	C			
process	B	E			

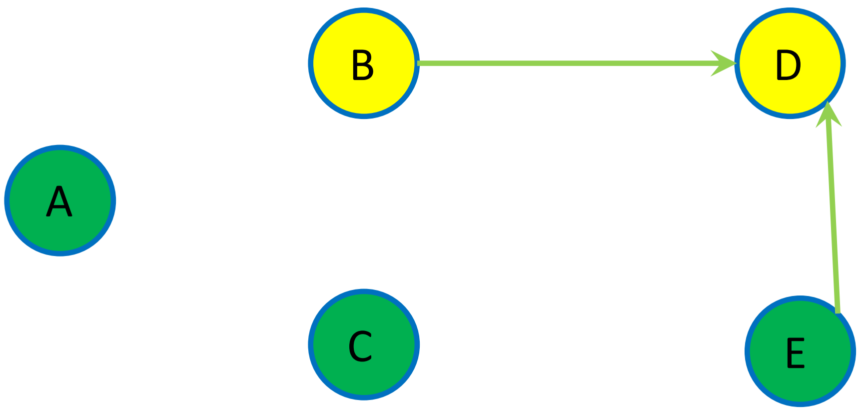
Kahn’s Algorithm
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



sorted	A	C	E		
process	B				

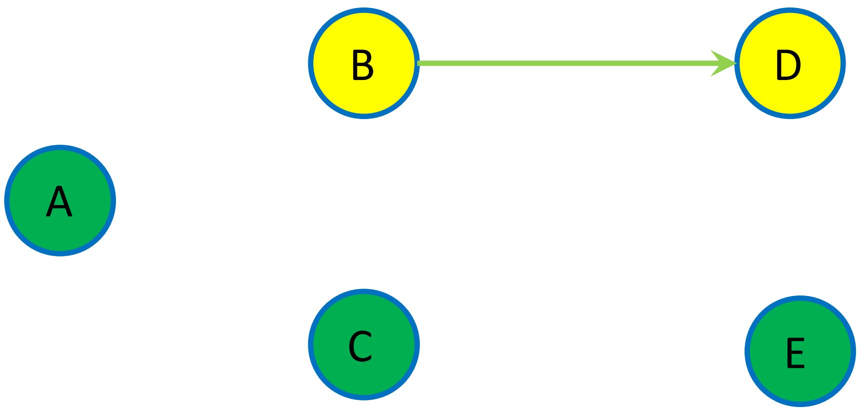
Kahn’s Algorithm
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



sorted	A	C	E		
process	B				

Kahn's Algorithm

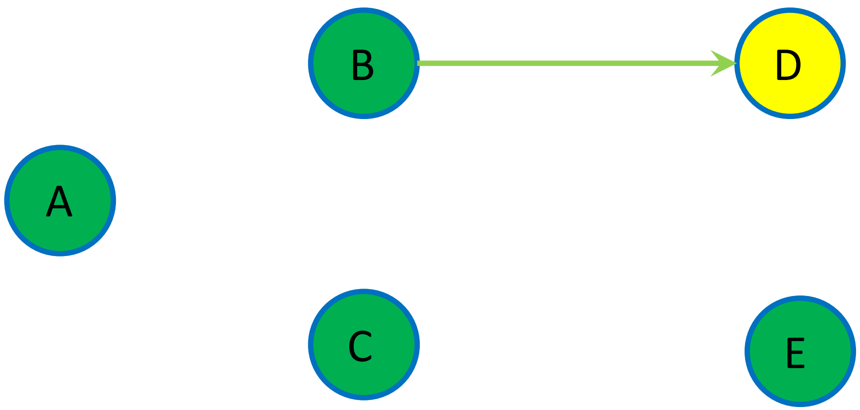
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



sorted	A	C	E	B	
process					

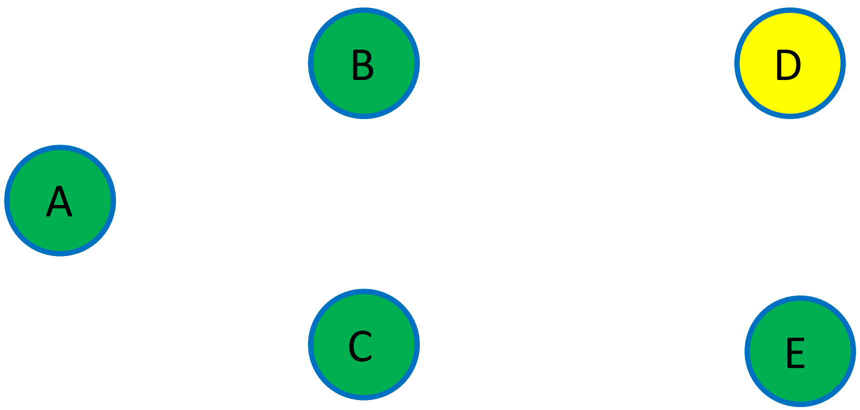
Kahn’s Algorithm
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



sorted	A	C	E	B	
process	D				

Kahn's Algorithm

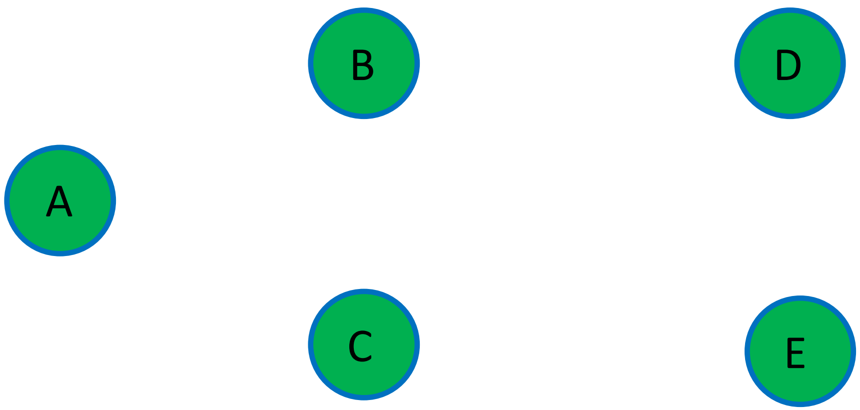
For topological sort

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

- Algorithm as follow
  - Let us try it out



not unique, can have different order of elements size of process > 1, then not unique

sorted	**A**	**C**	**E**	**B**	**D**
process					

choose either B or C to be with either queue or stack to be unique

Kahn’s Algorithm

For topological sort

- Algorithm as follow
  - Seemed simple right?
  - Let us zoomed in to the algorithm more

```
sorted_list = []
process = []
add all vertices without incoming edges to process

while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        remove edge from graph
        if edge.vertex_v has no incoming edges:
            process.append(vertex_v)

if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

Kahn’s Algorithm
For topological sort

- Algorithm as follow
  - Seemed simple right?
  - Let us zoomed in to the algorithm more

```python
# for output
sorted_list = []
# tracks number of incoming edges
incoming_edges = [0] * len(vertices)
for edge in graph:            # edge <u,v>
    incoming_edges[vertex_v] += 1
# process queue or stack
process = []
for vertex_v in incoming_edges:
    if incoming_edges[vertex_v] == 0:
        process.append(vertex_v)
# kahn's
while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        incoming_edges[edge.vertex_v] -= 1
        if incoming_edges[vertex_v] == 0:
            process.append(vertex_v)
# results
if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

Kahn's Algorithm

For topological sort

- Algorithm as follow
  - Seemed simple right?
  - Let us zoomed in to the algorithm more

- Complexity?

```
# for output
sorted_list = []
# tracks number of incoming edges
incoming_edges = [0] * len(vertices)
for edge in graph:          # edge <u,v>
    incoming_edges[vertex_v] += 1
# process queue or stack
process = []
for vertex_v in incoming_edges:
    if incoming_edges[vertex_v] == 0:
        process.append(vertex_v)
# kahn's
while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        incoming_edges[edge.vertex_v] -= 1
        if incoming_edges[vertex_v] == 0:
            process.append(vertex_v)
# results
if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

Kahn’s Algorithm ~= BFS

For topological sort

- Algorithm as follow
  - Seemed simple right?
  - Let us zoomed in to the algorithm more

O(E)

- Complexity?
  - O(V+E) time space

O(E)

```
# for output
sorted_list = []
# tracks number of incoming edges
incoming_edges = [0] * len(vertices)
for edge in graph:          # edge <u,v>
    incoming_edges[vertex_v] += 1
# process queue or stack
process = []
for vertex_v in incoming_edges:
    if incoming_edges[vertex_v] == 0:
        process.append(vertex_v)
# kahn's
while len(process) > 0:
    vertex_u = process.pop()
    sorted_list.append(vertex_u)
    for edge in vertex_u.edges:
        incoming_edges[edge.vertex_v] -= 1
        if incoming_edges[vertex_v] == 0:
            process.append(vertex_v)
# results
if graph still has edges:
    print("Error. Not a cycle")
else:
    print(sorted_list)
```

incoming edge don't care about u (start of edge)
only cares v (end of edge)

one vertex in process only has small part of all edges in graph
all these edges combine = O(E)
since process eventually run through all vertices
so O(V+E)

|incoming edges| > 0
still has edges
there was a cycle

Questions?

Depth-First Search (DFS)

Modified for topological sorting

- We saw the complexity of Kahn’s being O(V+E)

Depth-First Search (DFS)

Modified for topological sorting

- We saw the complexity of Kahn’s being O(V+E)
- Can we modify DFS to do the same?

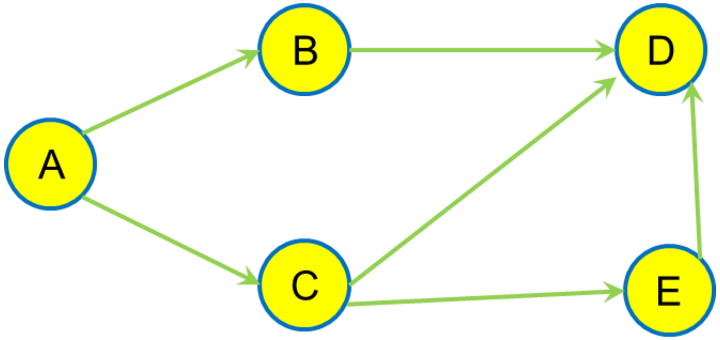
```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

Depth-First Search (DFS)

Modified for topological sorting

- Can we modify DFS to do the same?
  - Let us run DFS on the following graph and see

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

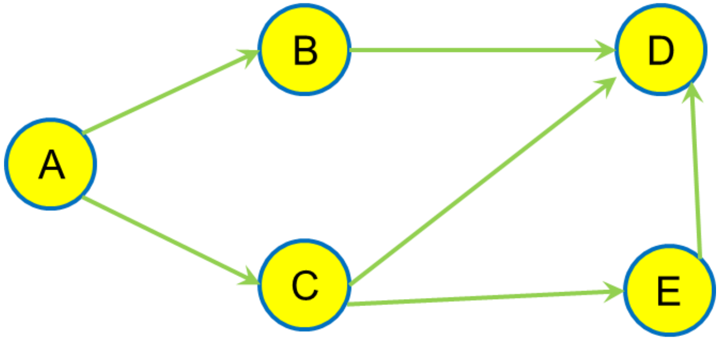


Depth-First Search (DFS)

Modified for topological sorting

- Can we modify DFS to do the same?
  - Let us run DFS on the following graph and see
    - Start from A

```
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

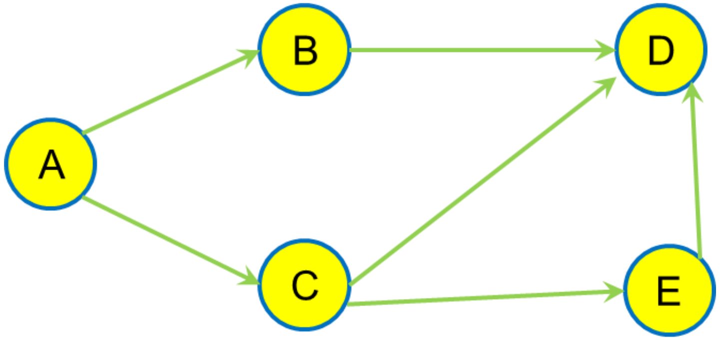


Depth-First Search (DFS)

Modified for topological sorting

- Can we modify DFS to do the same?
  - Let us run DFS on the following graph and see
    - Start from A
    - Go to B

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```



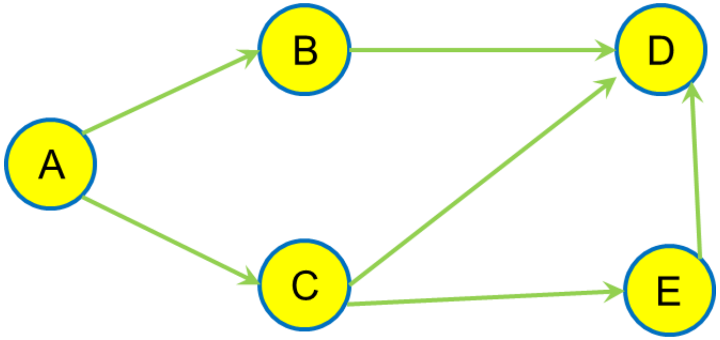
Depth-First Search (DFS)

Modified for topological sorting

MONASH University

- Can we modify DFS to do the same?
  - Let us run DFS on the following graph and see
    - Start from A
    - Go to B
    - Go to D

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```



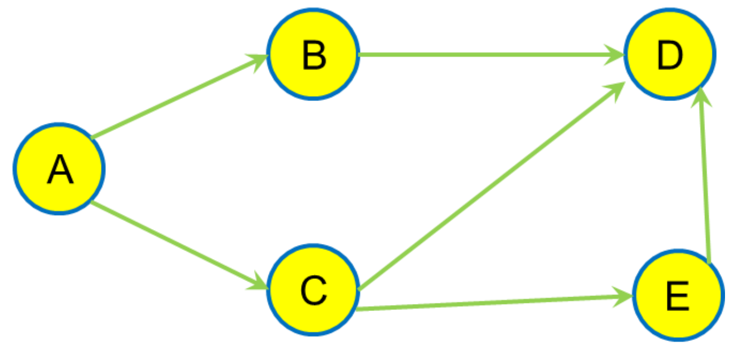
Depth-First Search (DFS)

Modified for topological sorting

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

- Can we modify DFS to do the same?
  - Let us run DFS on the following graph and see
    - Start from A
    - Go to B
    - Go to D
    - Go to C backtrack to A then go to C

recursion

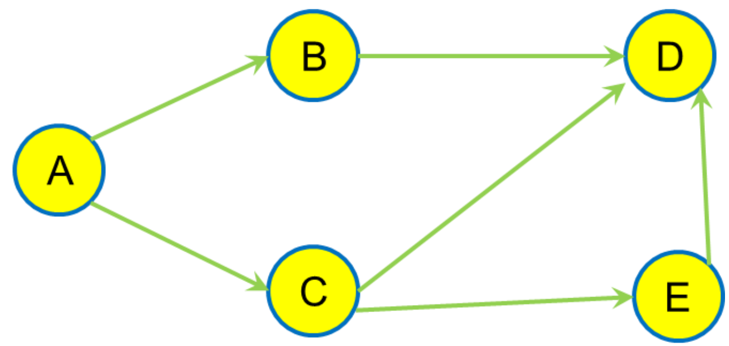


Depth-First Search (DFS)

Modified for topological sorting

```
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

- Can we modify DFS to do the same?
  - Let us run DFS on the following graph and see
    - Start from A
    - Go to B
    - Go to D
    - Go to C
    - Go to E



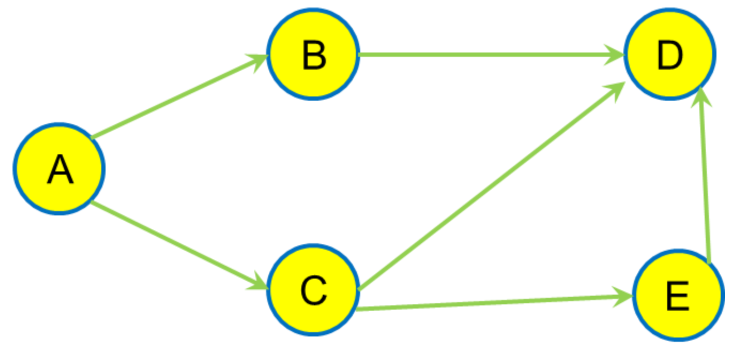
Depth-First Search (DFS)

Modified for topological sorting

MONASH University

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

- Can we modify DFS to do the same?
  - Let us run DFS on the following graph and see
    - Start from A
    - Go to B
    - Go to D
    - Go to C
    - Go to E
    - So we have A, B, D, C, E

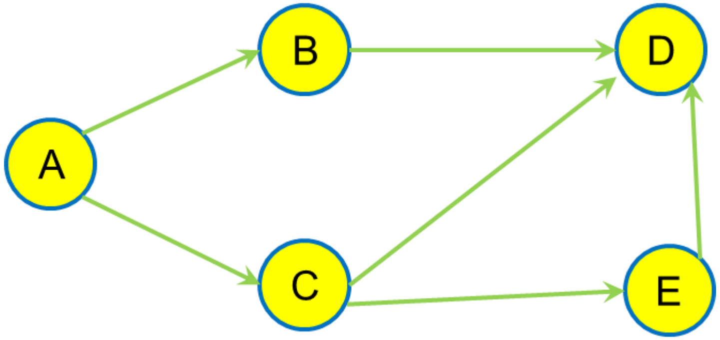


Depth-First Search (DFS)

Modified for topological sorting

- Can we modify DFS to do the same?
  - Let us run DFS on the following graph and see
    - Start from A
    - Go to B
    - Go to D
    - Go to C
    - Go to E
    - So we have A, B, D, C, E
  - Any other DFS order?

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

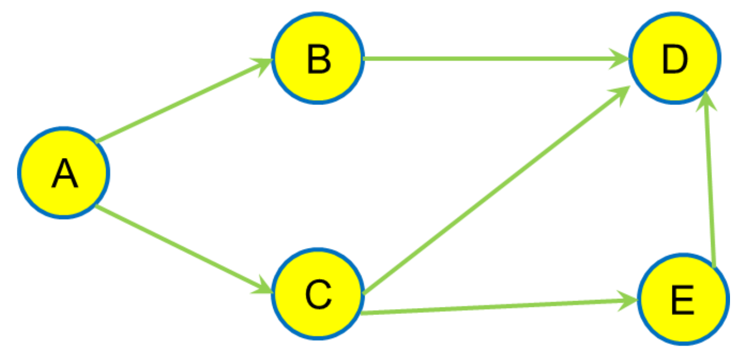


Depth-First Search (DFS)

Modified for topological sorting

- Can we modify DFS to do the same?
  - Let us run DFS on the following graph and see
    - Start from A
    - Go to B
    - Go to D
    - Go to C
    - Go to E
    - So we have A, B, D, C, E
  - Any other DFS order?
    - A, C, D, E, B

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

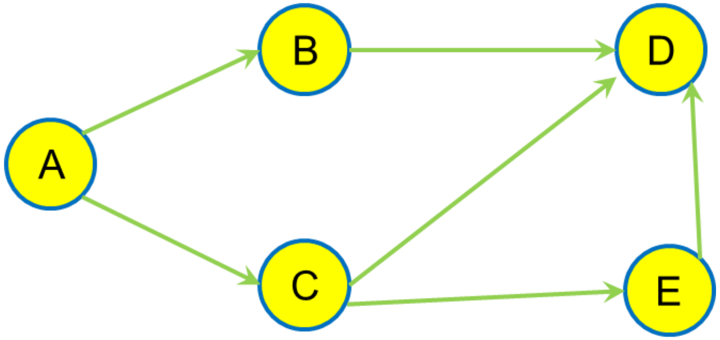


Depth-First Search (DFS)

Modified for topological sorting

- Can we modify DFS to do the same?
  - Let us run DFS on the following graph and see
    - Start from A
    - Go to B
    - Go to D
    - Go to C
    - Go to E
    - So we have A, B, D, C, E
  - Any other DFS order?
    - A, C, D, E, B
    - A, C, E, D, B

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

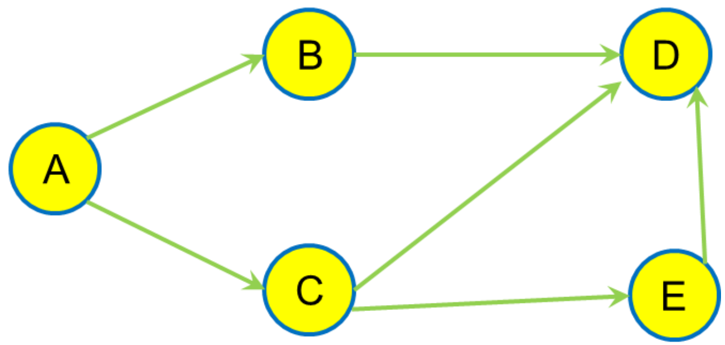


Depth-First Search (DFS)
Modified for topological sorting

MONASH University

- Can we modify DFS to do the same?
  - Let us run DFS on the following graph and see
    - Start from A
    - Go to B
    - Go to D
    - Go to C
    - Go to E
    - So we have A, B, D, C, E
  - Any other DFS order?
    - A, C, D, E, B
    - A, C, E, D, B
    - A, C, E, B, D?

```
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

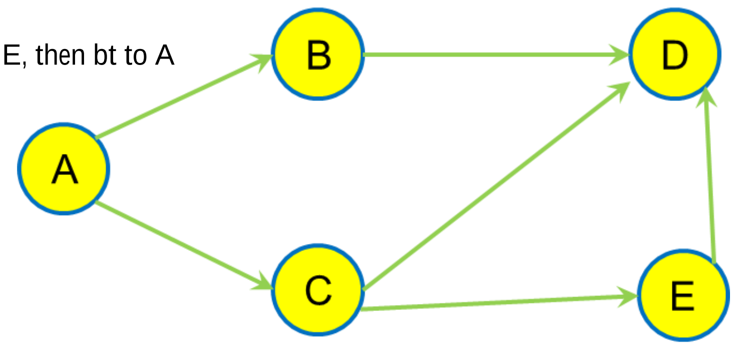


Depth-First Search (DFS)

Modified for topological sorting

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

- Can we modify DFS to do the same?
  - Any other DFS order?
    - A, B, D, C, E
    - A, C, D, E, B backtrack to C from D, then go to E, then bt to A then go to B
    - A, C, E, D, B

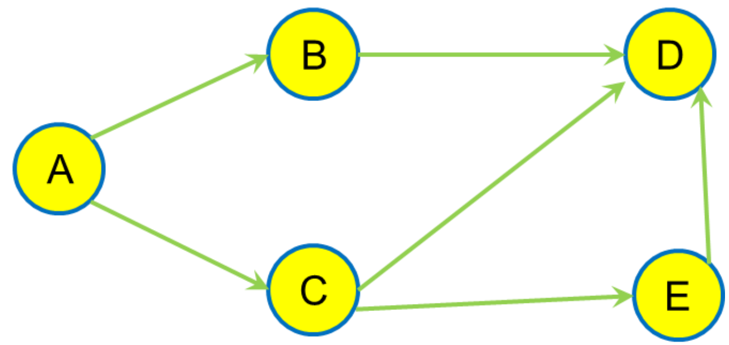


Depth-First Search (DFS)

Modified for topological sorting

- Can we modify DFS to do the same?
  - Any other DFS order?
    - A, B, D, C, E
    - A, C, D, E, B
    - A, C, E, D, B
  - A possible topological sort
    - A, B, C, E, D
    - A, C, B, E, D

```
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

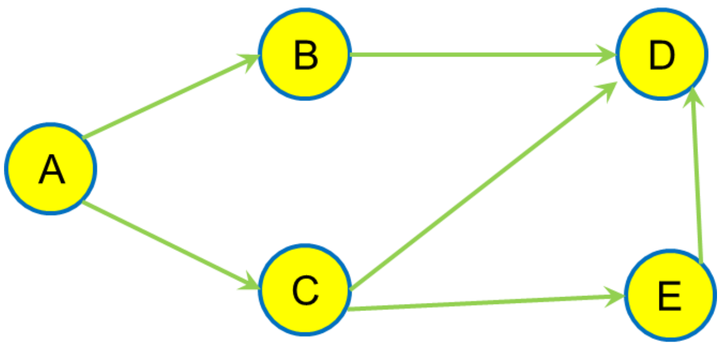


Depth-First Search (DFS)

Modified for topological sorting

- Can we modify DFS to do the same?
  - Any other DFS order?
    - A, B, D, C, E
    - A, C, D, E, B
    - A, C, E, D, B
  - A possible topological sort
    - A, B, C, E, D
    - A, C, B, E, D
  - Notice something?

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```



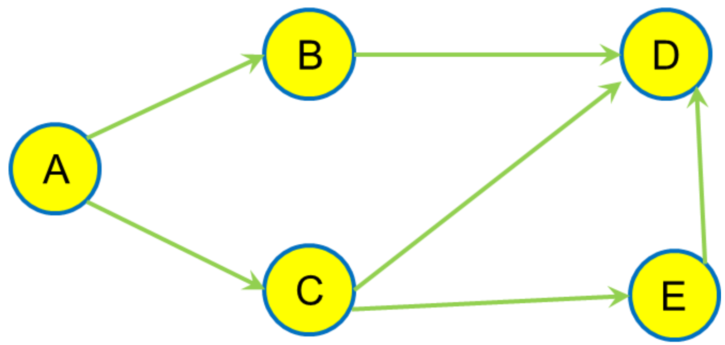
Depth-First Search (DFS)

Modified for topological sorting

MONASH University

- Can we modify DFS to do the same?
  - Any other DFS order?
    - A, B, D, C, E
    - A, C, D, E, B
    - A, C, E, D, B
  - A possible topological sort
    - A, B, C, E, D
    - A, C, B, E, D
  - Notice something?
    - When we reach the end of the DFS, we go back to an earlier vertex but this vertex should be early in topological sort (such as vertex B or C)

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

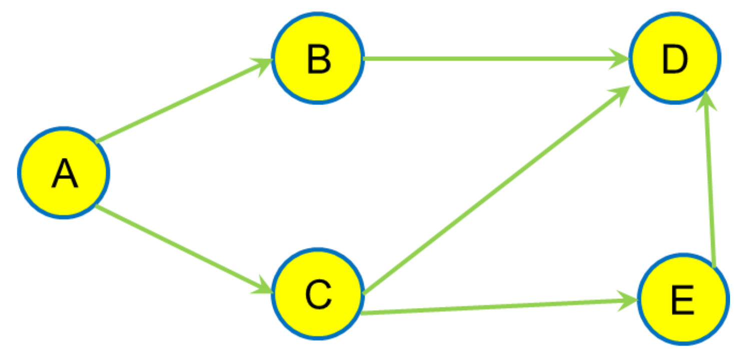


Depth-First Search (DFS)

Modified for topological sorting

```python
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

- Can we modify DFS to do the same?
  - Any other DFS order?
    - A, B, D, C, E
    - A, C, D, E, B
    - A, C, E, D, B
  - A possible topological sort
    - A, B, C, E, D
    - A, C, B, E, D
  - Notice something?
    - When we reach the end of the DFS, we go back to an earlier vertex but this vertex should be early in topological sort (such as vertex B or C)

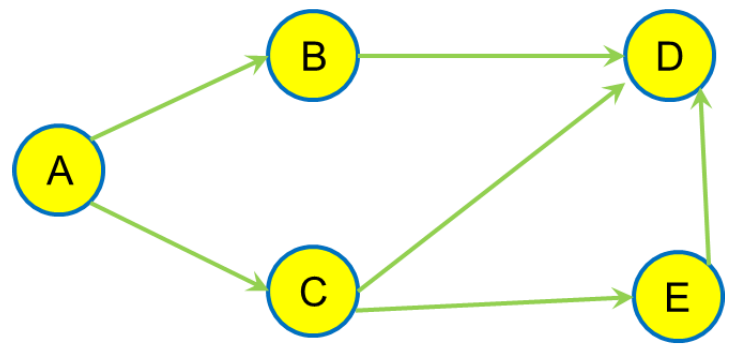


Depth-First Search (DFS)

Modified for topological sorting

```
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

- Can we modify DFS to do the same?
  - Any other DFS order?
    - A, B, D, C, E
    - A, C, D, E, B
    - A, C, E, D, B
  - A possible topological sort
    - A, B, C, E, D
    - A, C, B, E, D
  - Notice something?
    - When we reach the end of the DFS, we go back to an earlier vertex but this vertex should be early in topological sort (such as vertex B or C)



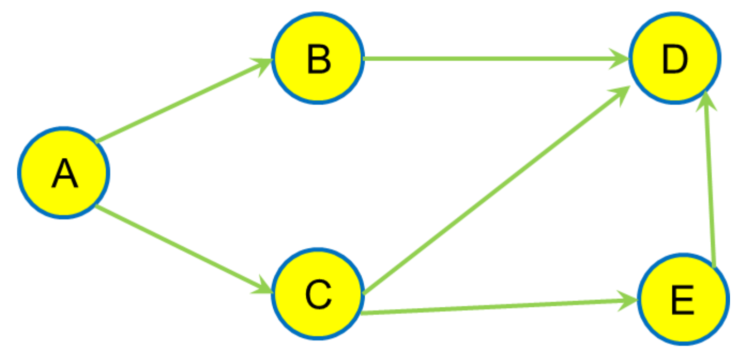
Depth-First Search (DFS)

Modified for topological sorting

MONASH University

```
def dfs_topological(vertex_u):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological(vertex_v)
```

- Can we modify DFS to do the same?
  - Any other DFS order?
    - A, B, D, C, E <- push D, B then Push E, C finally A
    - A, C, D, E, B
    - A, C, E, D, B original： A,B,C,D,E
  - A possible topological sort
    - A, B, C, E, D for modified DFS
    - A, C, B, E, D for BFS
  - Notice something?
    - When we reach the end of the DFS, we go back to an earlier vertex but this vertex should be early in topological sort (such as vertex B or C)
    - So how do we arrange it?



Depth-First Search (DFS)
Modified for topological sorting

- Here’s how we modify with a stack!

last in first out

```python
def dfs_topological(vertex_u):
    # start for result
    stack = []
    # run DFS
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological_aux(vertex_v,stack)
    # output
    print(stack)

def dfs_topological_aux(vertex_u,stack):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological_aux(vertex_v)
    # add to stack
    stack.push(vertex_u)
```

backtrack to here to push to stack

separate into two, make sure A(root) to be pushed to stack

DFS

edge.vertex_v = D == True

only do it on A, C,E,D, B
push D, E, C, B, A
serve A, B, C, E, D

Depth-First Search (DFS)

Modified for topological sorting

- Complexity?

```python
def dfs_topological(vertex_u):
    # start for result
    stack = []
    # run DFS
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological_aux(vertex_v,stack)
    # output
    print(stack)

def dfs_topological_aux(vertex_u,stack):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological_aux(vertex_v)
    # add to stack
    stack.push(vertex_u)
```

not next: just mark edge.vertex_v = TRUE

reverse the stack on purpose
since it first in last out

push back in reverse order since
visit: A -> B -> D -> C -> E ->
push D -> B -> E -> C -> A
serve

Depth-First Search (DFS)

Modified for topological sorting

- Complexity?
  - O(V+E) since we only added a stack

```
def dfs_topological(vertex_u):
    # start for result
    stack = []
    # run DFS
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological_aux(vertex_v,stack)
    # output
    print(stack)

def dfs_topological_aux(vertex_u,stack):
    vertex_u.visited = True
    for edge in vertex_u.edges:
        if edge.vertex_v.visited == False:
            dfs_topological_aux(vertex_v)
    # add to stack
    stack.push(vertex_u)
```

Questions?

Thank You