COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 15: Graph DFS + Topological Sort

Lecturer: Chris Lucas (cflucas@ncat.edu)

Midterm Grades Released!

Midpoint Grades Submitted!

HW4 grades by EoW!

HW5 due in 1 week!

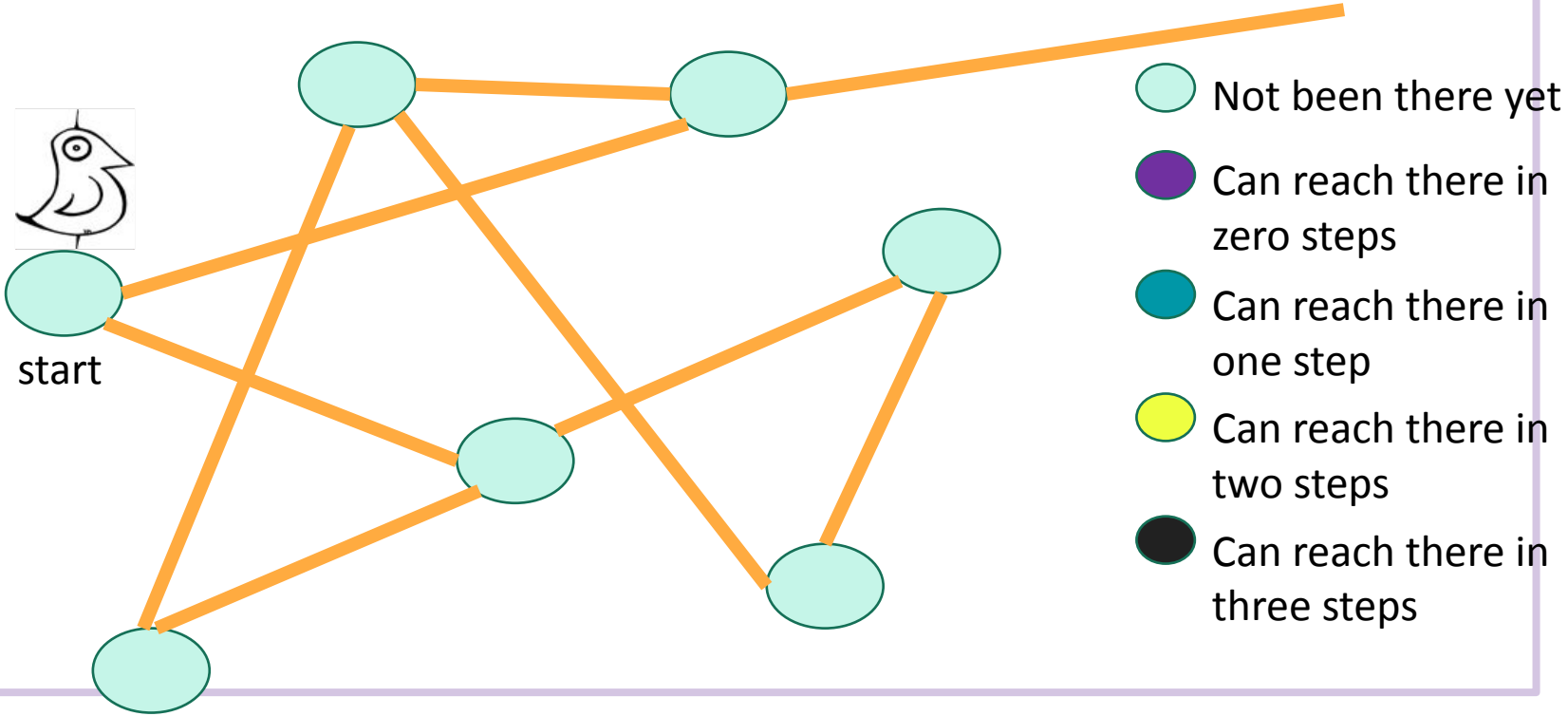
10/25 @ 11:59PM ET

Quiz!

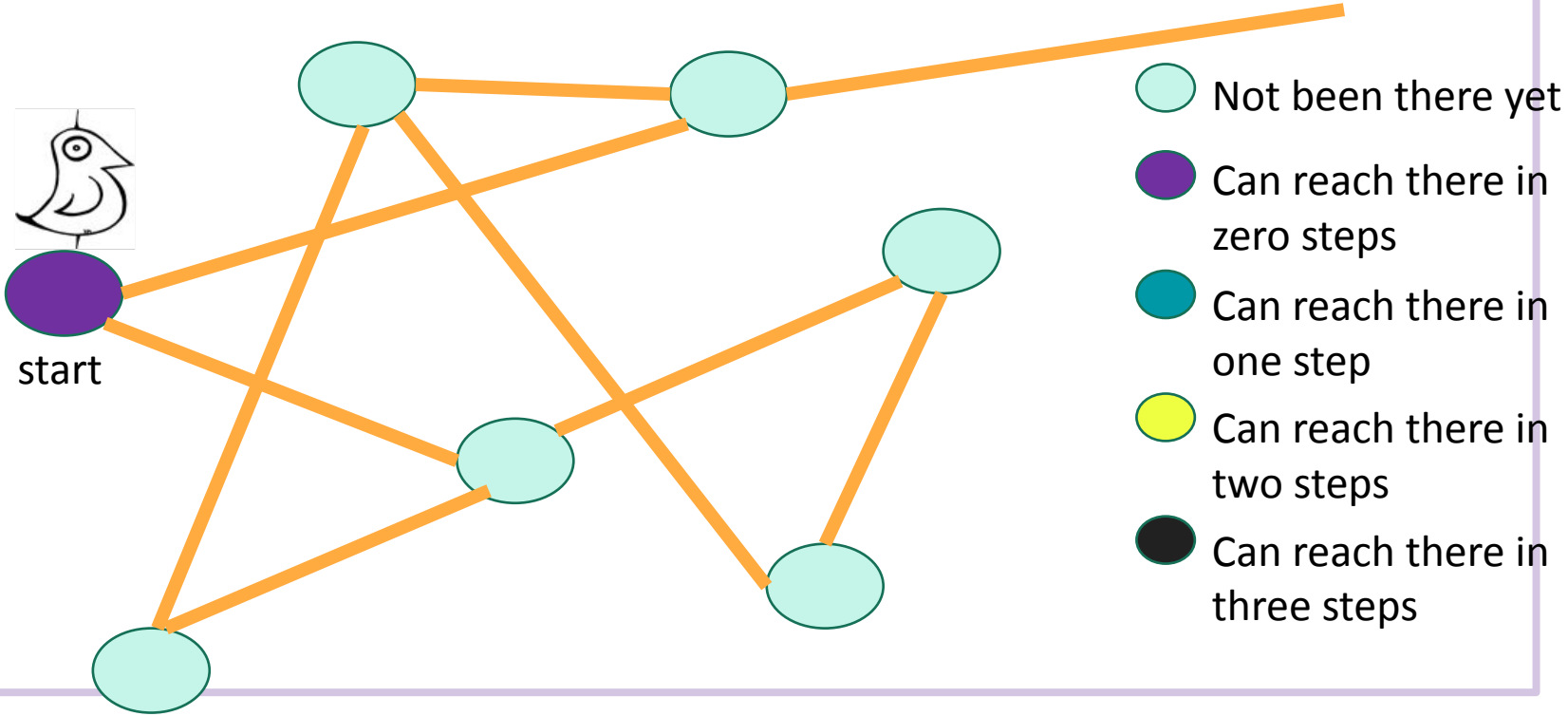
www.comp285-fall22.ml

Recall where we ended last lecture…

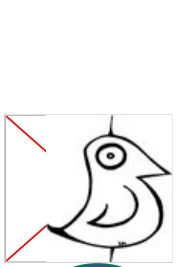
Breadth-First Search: Exploring the world with a bird's-eye view



Breadth-First Search: Exploring the world with a bird's-eye view



Breadth-First Search: Exploring the world with a bird's-eye view



start

Not been there yet

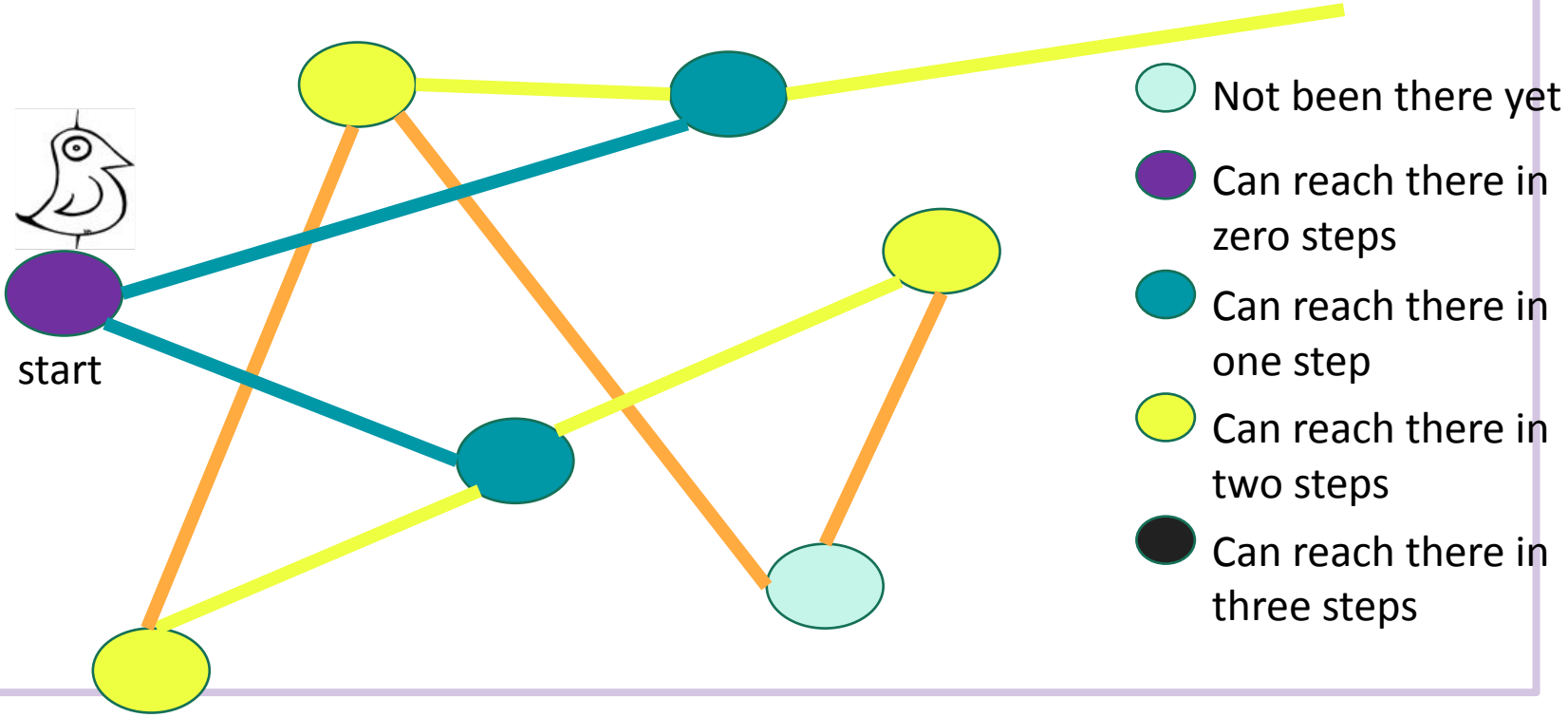
Can reach there in zero steps

Can reach there in one step

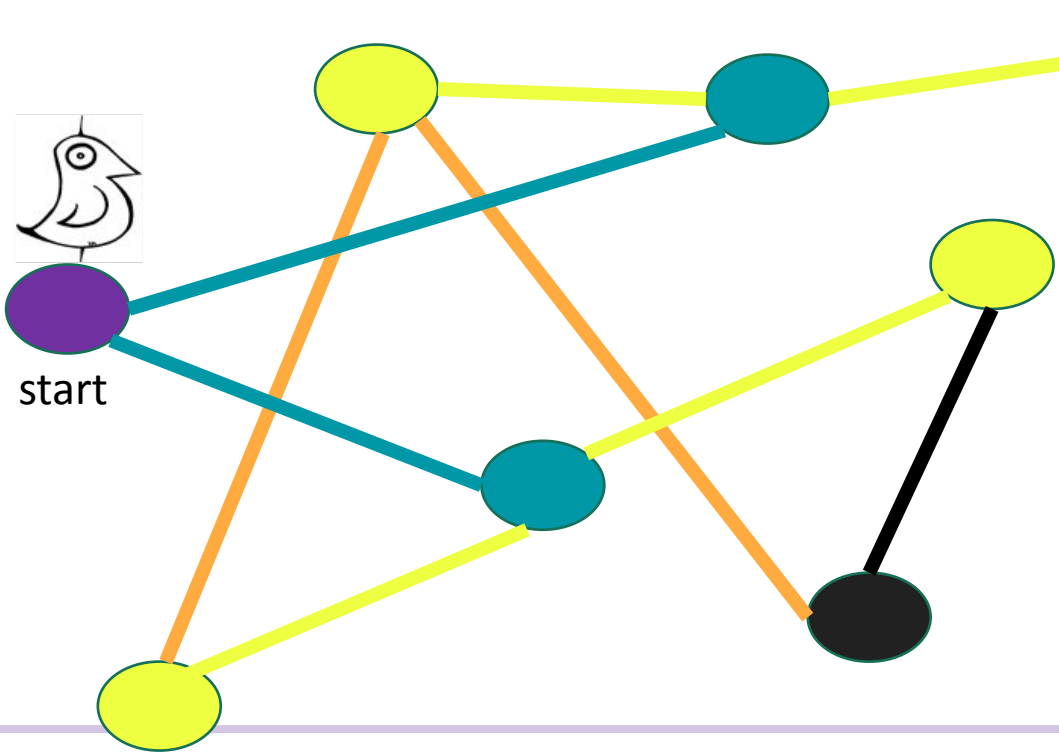
Can reach there in two steps

Can reach there in three steps

Breadth-First Search: Exploring the world with a bird's-eye view



Breadth-First Search: Exploring the world with a bird's-eye view



Not been there yet

Can reach there in zero steps

Can reach there in one step

Can reach there in two steps

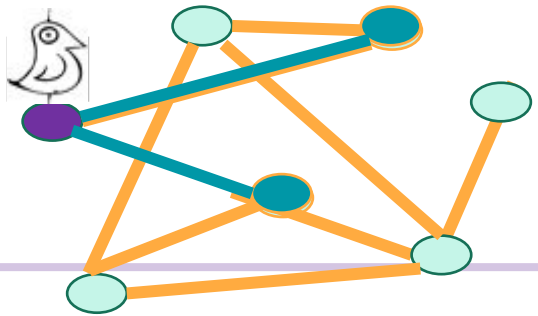
Can reach there in three steps

World:

explored!

Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), s and d
    Output: true/false if path from s to d

    frontier = Queue of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

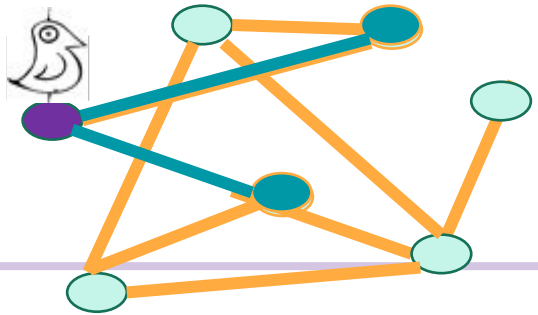


Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), s and d
    Output: true/false if path from s to d

    frontier = Queue of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

Runtime: O(V + E)

Why? What does this signify?

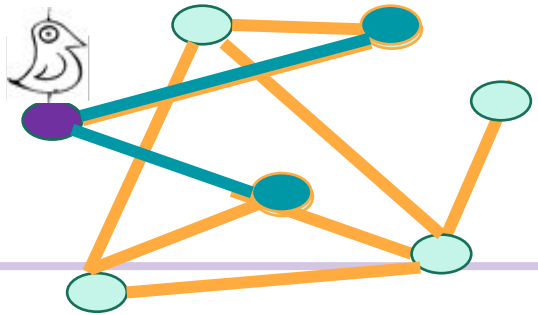


Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), s and d
    Output: true/false if path from s to d

    frontier = Queue of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

Runtime: O(V + E)

Traversing every edge and vertex!

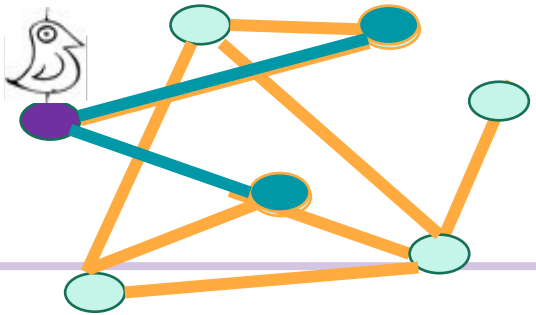


Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), s and d
    Output: true/false if path from s to d

    frontier = Queue of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

Runtime: O(V + E)

Space Complexity: ???

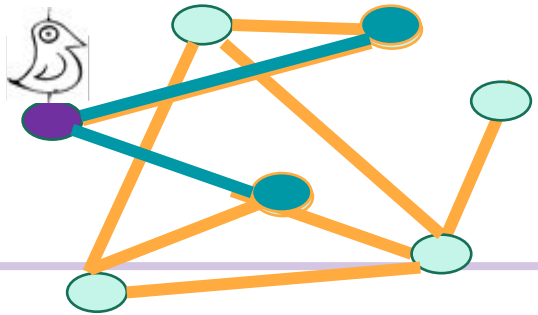


Breadth-First Search Pseudocode

```
algorithm bfs
    Input: undirected graph G = (V,E), s and d
    Output: true/false if path from s to d

    frontier = Queue of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

Runtime: O(V + E)

Space Complexity: O(V)



Summary of Breadth-First Search

- BFS can be written such that starting from a node v:
  - You find a path to every other vertex in G (if one exists)
  - You find a path to a specific destination node w
  - (Depends on your use case)
- BFS is guaranteed to find the shortest path between nodes
- BFS running time is O(V + E): traversing every node and every edge
- Note that BFS also works on directed graphs.

Big Questions!

- What about Depth-first search (DFS)?
- What's a DAG?
- What about topological sorting (TopoSort)?

Big Questions!

- What about Depth-first search (DFS)?
- What's a DAG?
- What about topological sorting (TopoSort)?

Analogy

Analogy

BFS

Analogy

BFS

DFS

Breadth/Depth-First Search Pseudocode

```
algorithm BFS
   Input: undirected graph G = (V,E), s and d
   Output: true/false if path from s to d

    frontier = Queue of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

Breadth/Depth-First Search Pseudocode

```
algorithm BFS
   Input: undirected graph G = (V,E), s and d
   Output: true/false if path from s to d

    frontier = Queue of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

```
algorithm DFS
   Input: undirected graph G = (V,E), s and d
   Output: true/false if path from s to d

    frontier = Stack of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

https://visualgo.net/en/dfsbfs

One More DFS Implementation…

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

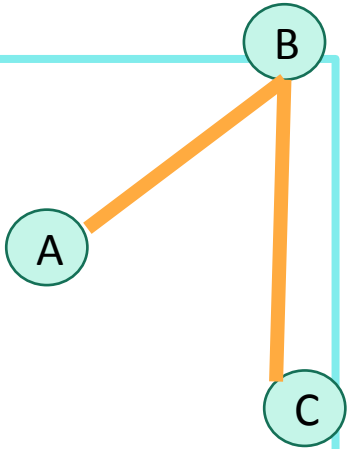
```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```

One More DFS Implementation...

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

DFS(A, C) ?



algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```

One More DFS Implementation...

algorithm DFS

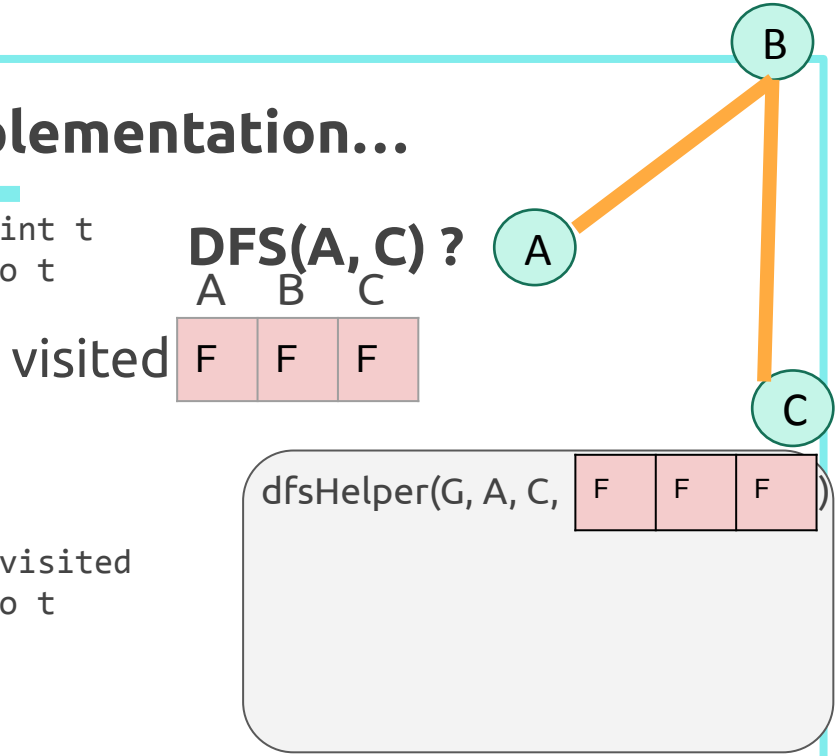
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

```
visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

```
visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```



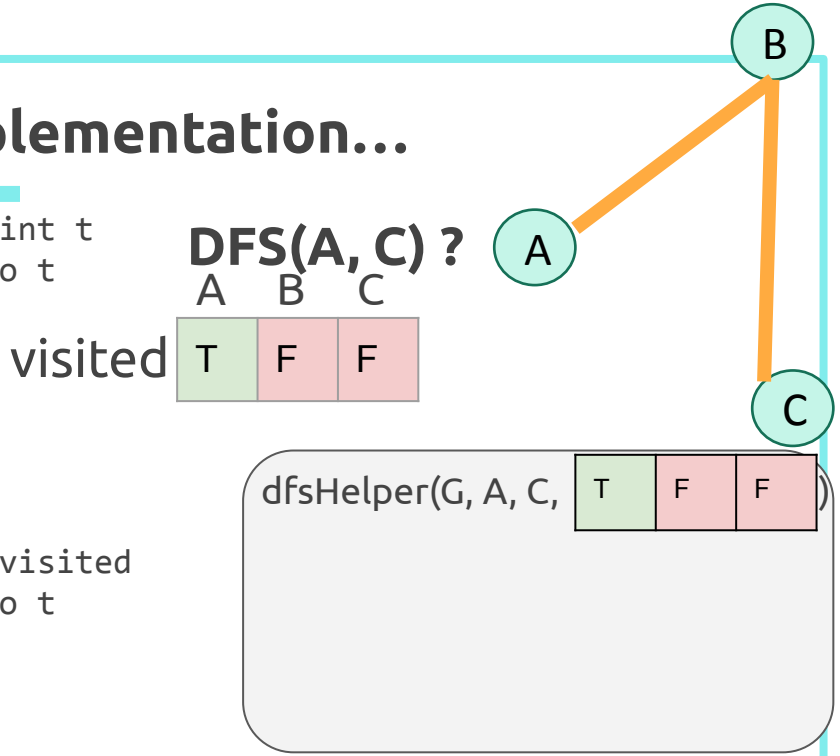
One More DFS Implementation...

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```



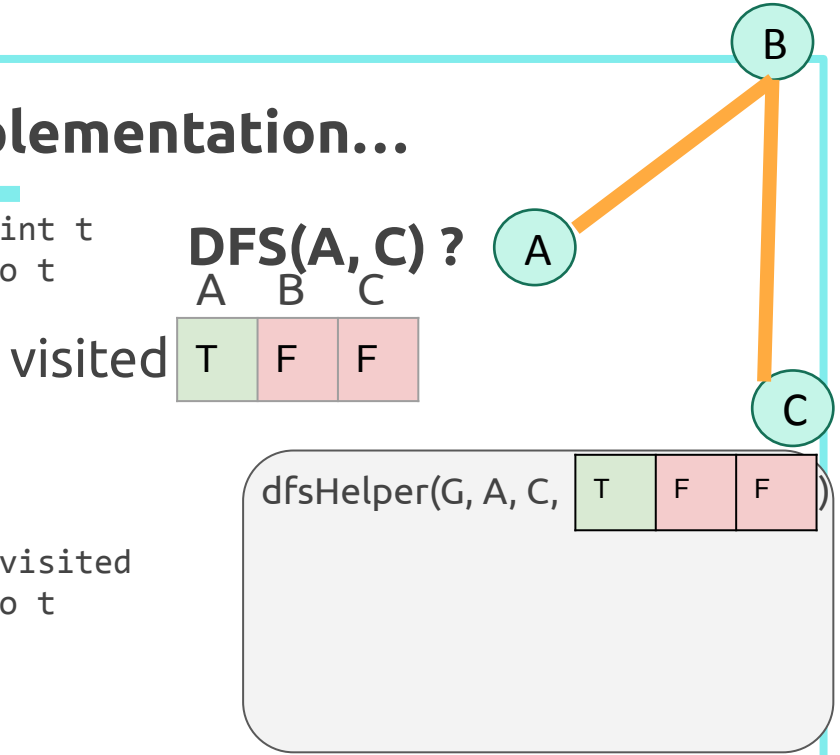
One More DFS Implementation...

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```



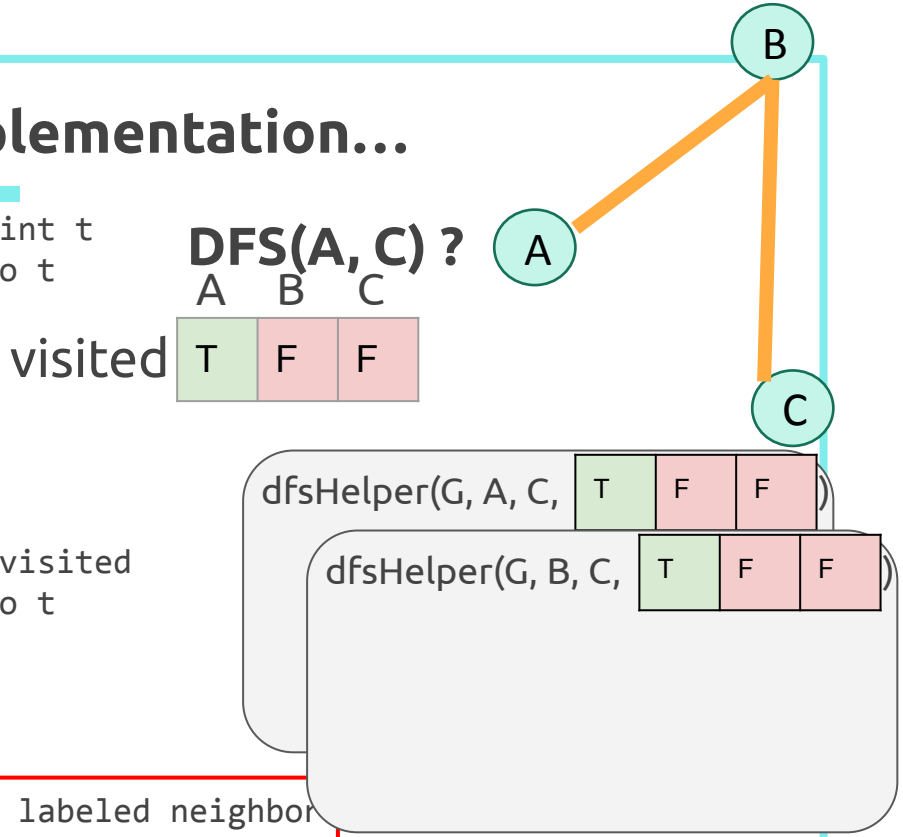
One More DFS Implementation...

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```



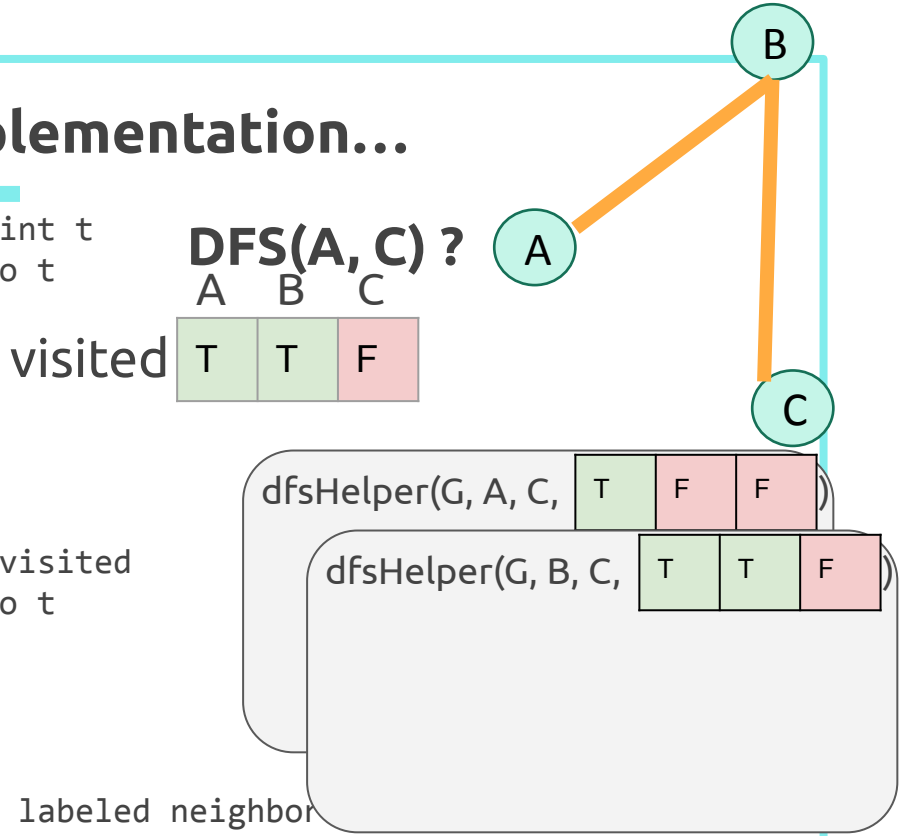
One More DFS Implementation...

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```



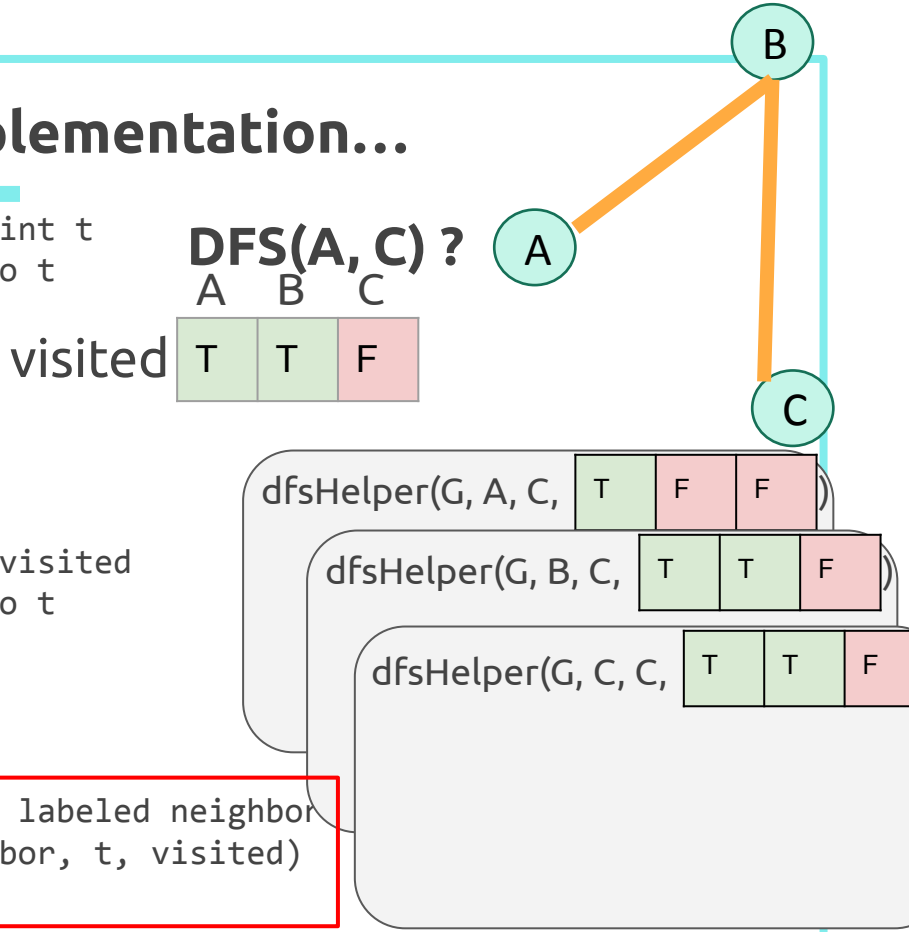
One More DFS Implementation...

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```



One More DFS Implementation...

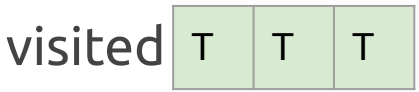
algorithm DFS

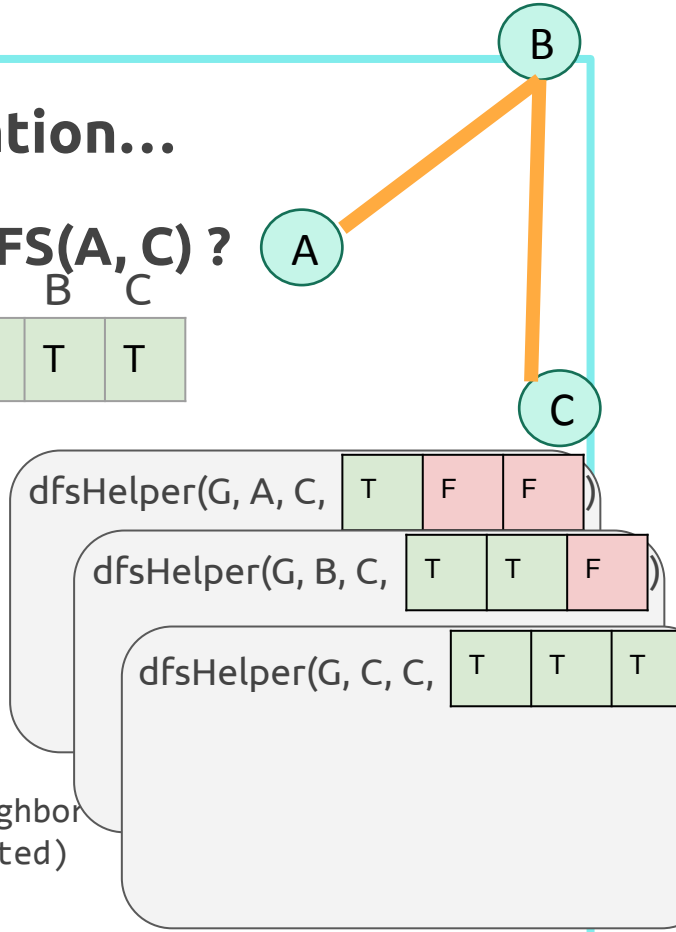
```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```

DFS(A, C) ?





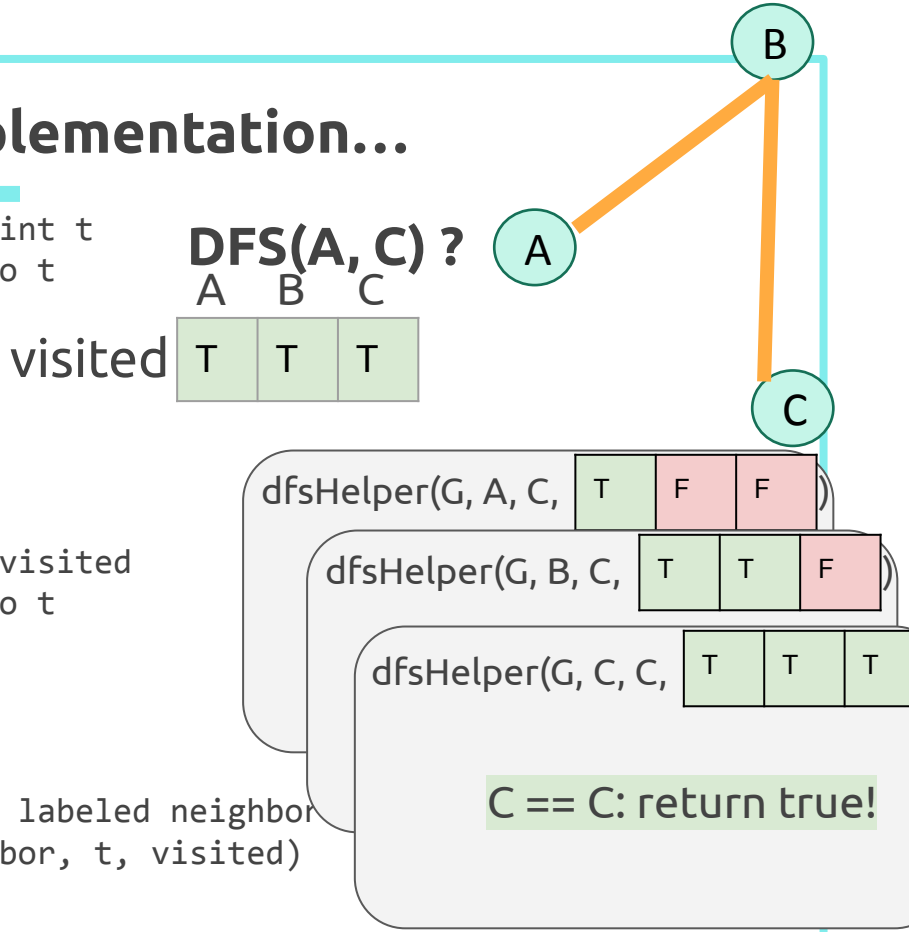
One More DFS Implementation...

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```



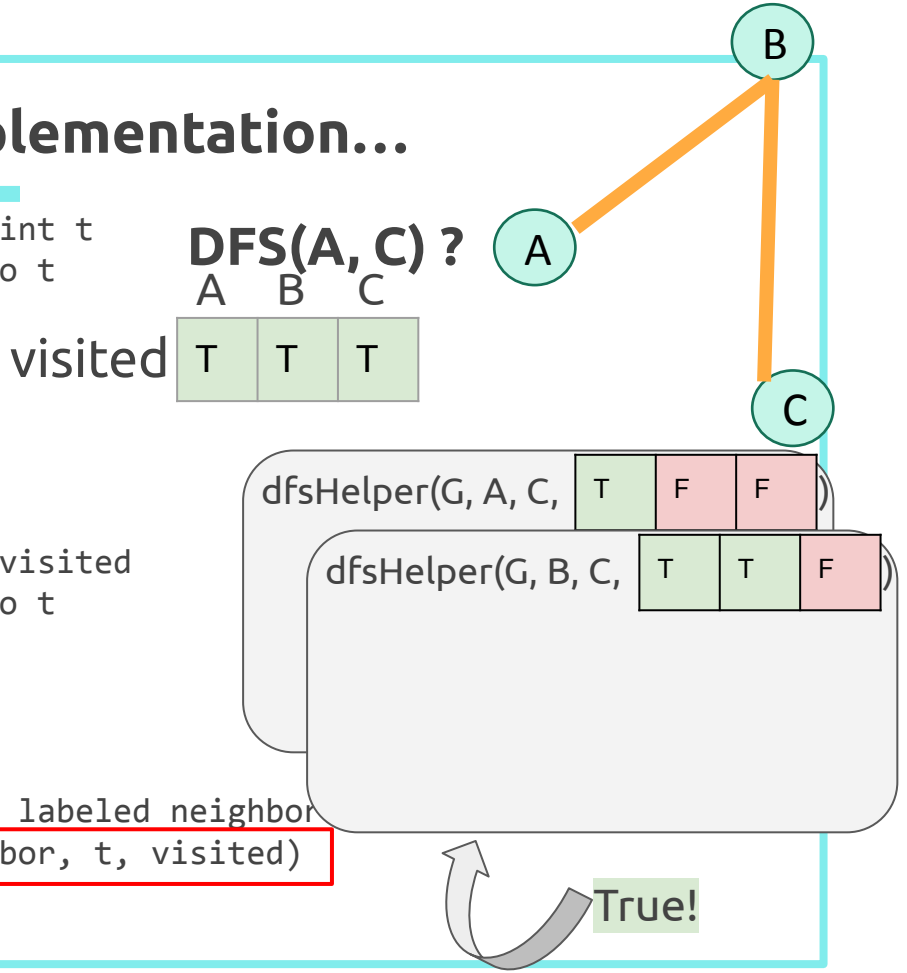
One More DFS Implementation...

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```



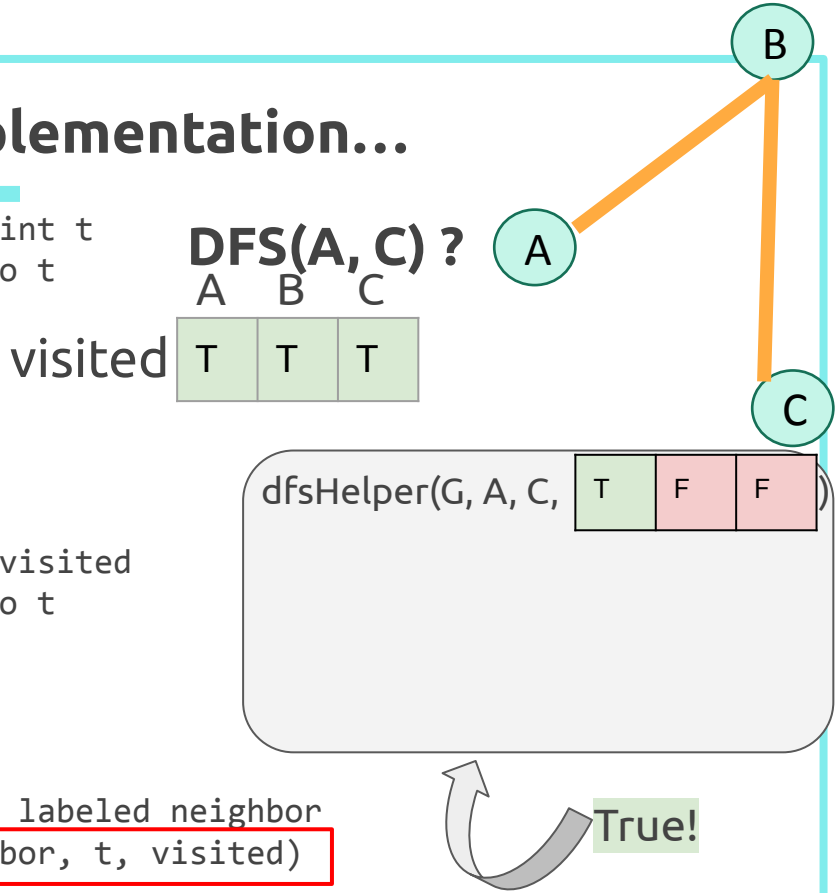
One More DFS Implementation...

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```



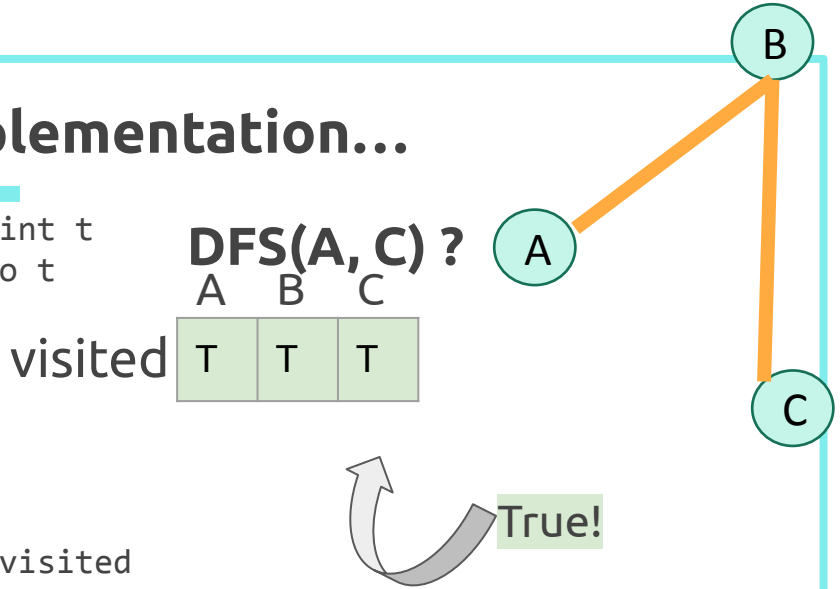
One More DFS Implementation...

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```



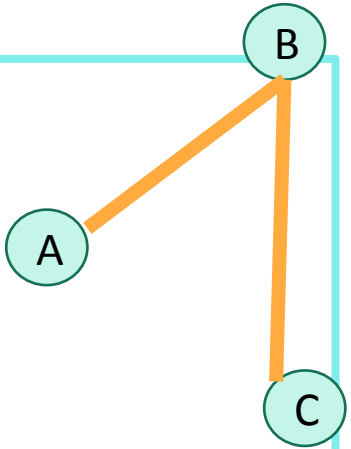
One More DFS Implementation...

algorithm DFS

```
Input: undirected graph G = (V, E), int s and int t
Output: whether or not there's a path from s to t

visited = new boolean array of size |V|
return dfsHelper(G, s, t, visited)
```

algorithm DFSHelper

```
Input: undirected graph G = (V, E), s, t, and visited
Output: whether or not there's a path from s to t

visited[s] = true
if s == t
  return true
for each neighbor of s, starting from smallest labeled neighbor
  if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
    return true
return false
```



DFS(A, C) is True!

Kahoot!

www.kahoot.it, Code: XXX YYY

Enter your @aggies.ncat email

One More DFS Implementation…

```
algorithm DFS
 Input: undirected graph G = (V, E), int s and int t
 Output: whether or not there's a path from s to t

 visited = new boolean array of size |V|
 return dfsHelper(G, s, t, visited)
```

Time Complexity? O(V + E)
Space Complexity? O(V)

```
algorithm DFSHelper
 Input: undirected graph G = (V, E), s, t, and visited
 Output: whether or not there's a path from s to t

 visited[s] = true
 if s == t
   return true
 for each neighbor of s, starting from smallest labeled neighbor
   if !visited[neighbor] and dfsHelper(G, neighbor, t, visited)
     return true
 return false
```

Depth-First Search Pseudocode

```
algorithm DFS
 Input: undirected graph G, int s and int t
 Output: whether or not there's a path from s to t

 visited = new boolean array of size |V|
 return dfsHelper(G, s, t, visited)


algorithm DFSHelper
 Input: undirected graph G = (V, E), s, t, and
visited
 Output: whether or not there's a path from s to t

 visited[s] = true
 if s == t
   return true
 for each neighbor of s, starting from smallest
labeled neighbor
   if !visited[neighbor] and dfsHelper(G,
neighbor, t, visited)
     return true
 return false
```

```
algorithm DFS
   Input: undirected graph G = (V,E), s and d
   Output: true/false if path from s to d

    frontier = Stack of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

Depth-First Search Pseudocode

```
algorithm DFS
 Input: undirected graph G, int s and int t
 Output: whether or not there's a path from s to t

 visited = new boolean array of size |V|
 return dfsHelper(G, s, t, visited)


algorithm DFSHelper
 Input: undirected graph G = (V, E), s, t, and
visited
 Output: whether or not there's a path from s to t

 visited[s] = true
 if s == t
   return true
 for each neighbor of s, starting from smallest
labeled neighbor
   if !visited[neighbor] and dfsHelper(G,
neighbor, t, visited)
     return true
 return false
```

Time Complexity? O(V + E)
Space Complexity? O(V)

```
algorithm DFS
   Input: undirected graph G = (V,E), s and d
   Output: true/false if path from s to d

    frontier = Stack of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

Time Complexity? ???
Space Complexity? ???

Kahoot!

www.kahoot.it, Code: XXX YYY

Enter your @aggies.ncat email

Depth-First Search Pseudocode

```
algorithm DFS
 Input: undirected graph G, int s and int t
 Output: whether or not there's a path from s to t

 visited = new boolean array of size |V|
 return dfsHelper(G, s, t, visited)


algorithm DFSHelper
 Input: undirected graph G = (V, E), s, t, and
visited
 Output: whether or not there's a path from s to t

 visited[s] = true
 if s == t
   return true
 for each neighbor of s, starting from smallest
labeled neighbor
   if !visited[neighbor] and dfsHelper(G,
neighbor, t, visited)
     return true
 return false
```

Time Complexity? O(V + E)
Space Complexity? O(V)

```
algorithm DFS
   Input: undirected graph G = (V,E), s and d
   Output: true/false if path from s to d

    frontier = Stack of integers
    visited = {} // empty hash set
    frontier.add(s)
    visited.insert(s)
    while not frontier.empty()
      currNode = frontier.remove()
      if currNode == d
        return true
      for each neighbor of currNode
        if neighbor not in visited
          visited.insert(neighbor)
          frontier.add(neighbor)
    return false
```

Time Complexity? O(V+E)
Space Complexity? O(V)

Application: Flood Fill

- Microsoft Paint Bucket Tool
- Note that images can be represented as a grid of pixels
- To fill every connected pixel that is the same color as the “seed”, we can use a BFS or DFS
- We look up, down, left, and right, and explore the next node only if it is the same color as the seed

Big Questions!

- What about Depth-first search (DFS)?
- What's a DAG?
- What about topological sorting (TopoSort)?

Directed Acyclic Graphs (DAGs)

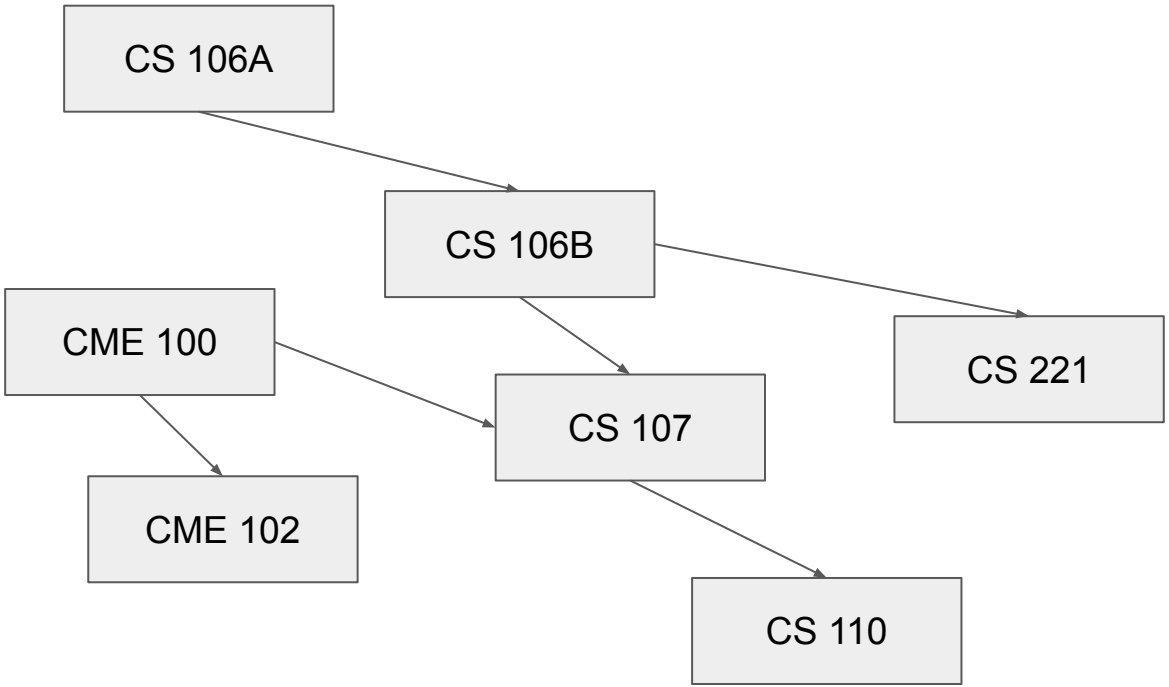
When we have a graph that is both directed and has no loops (acyclic), we call it a Directed Acyclic Graph (a.k.a DAG).

They’re useful for representing real-world dependencies.

Directed graphs (not just DAGs) have nodes that are:

- Sources: no inbound edges (i.e. no arrows pointing to it)
- Sinks: no outbound edges (i.e. no arrows pointing away from it)

Course Prerequisites



Kahoot!

www.kahoot.it, Code: XXX YYYY

Enter your @aggies.ncat email

Polls

Is this a DAG?

Yes

Which Nodes are source nodes?

3, 5

Which Nodes are sink nodes?

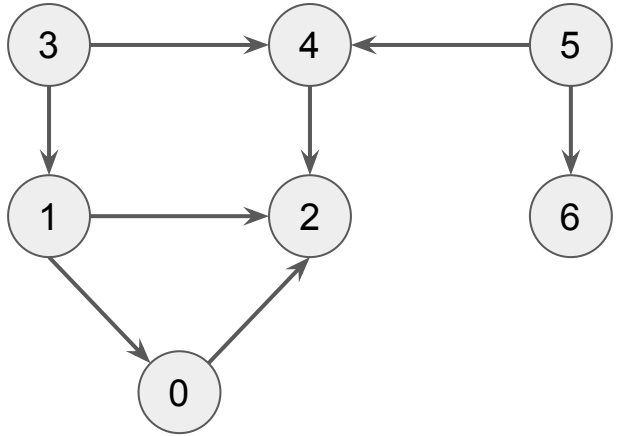
2, 6

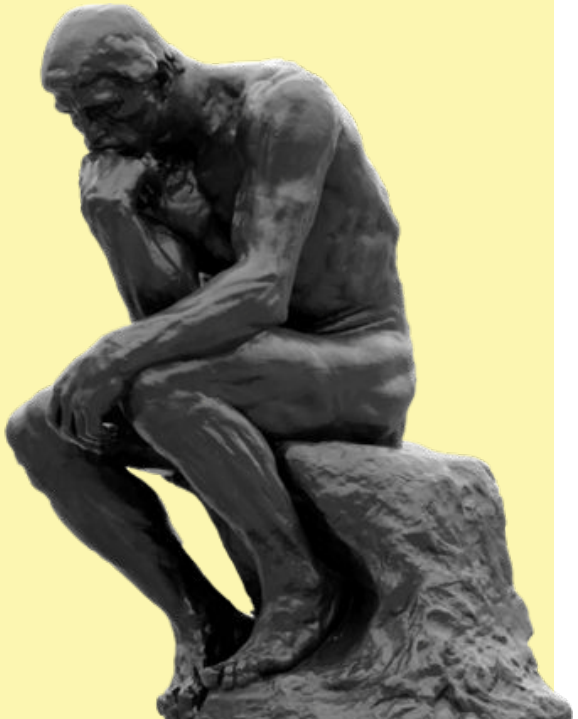
If we ran DFS starting at 4 and followed only the outbound edges, what would we print out?

4, 2

If we ran DFS starting at 3 and only followed the outbound edges, what would we print out? (Ties broken numerically, smallest node value first)

3, 1, 0, 2, 4





Big Questions!

- What about Depth-first search (DFS)?
- What's a DAG?
- What about topological sorting (TopoSort)?

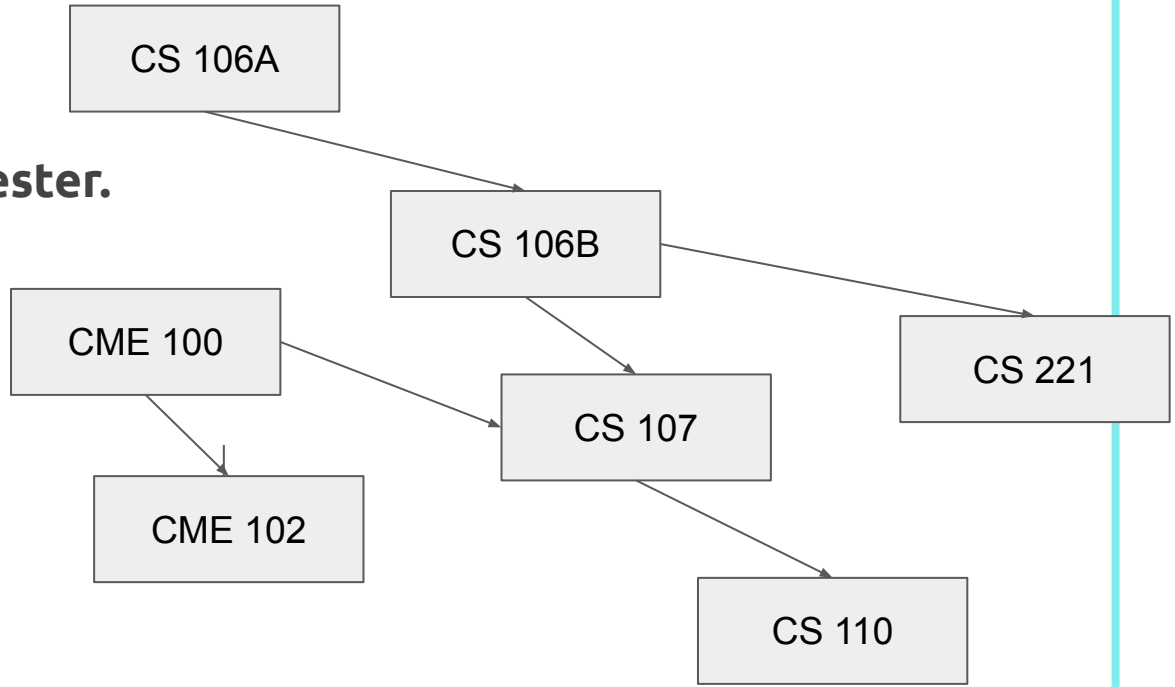
Topological Sort

- "A topological sort of a DAG G = (V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in the ordering. (If the graph contains a cycle,then no linear ordering is possible.)"
  - Fancy way of saying “give me one valid ordering of a DAG’s nodes.”
  - If U -> V in graph, means node U must occur before node V in order
  - Only possible to calculate this for DAGs
- A valid topological sort will have all "pre-req" nodes appear before the nodes that depend on those pre-reqs.
- "Sort" is misleading. Don't confuse this with the sorts we saw before. Topological “ordering” is another name that may be less confusing.

Topological Sort Example #1

What's one valid course ordering?
Assume 1 course per semester.

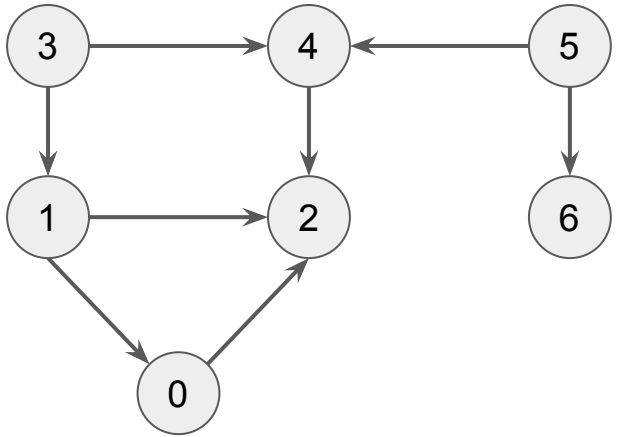
CME 100
CME 102
CS 106A
CS 106B
CS 107
CS 221
CS 110



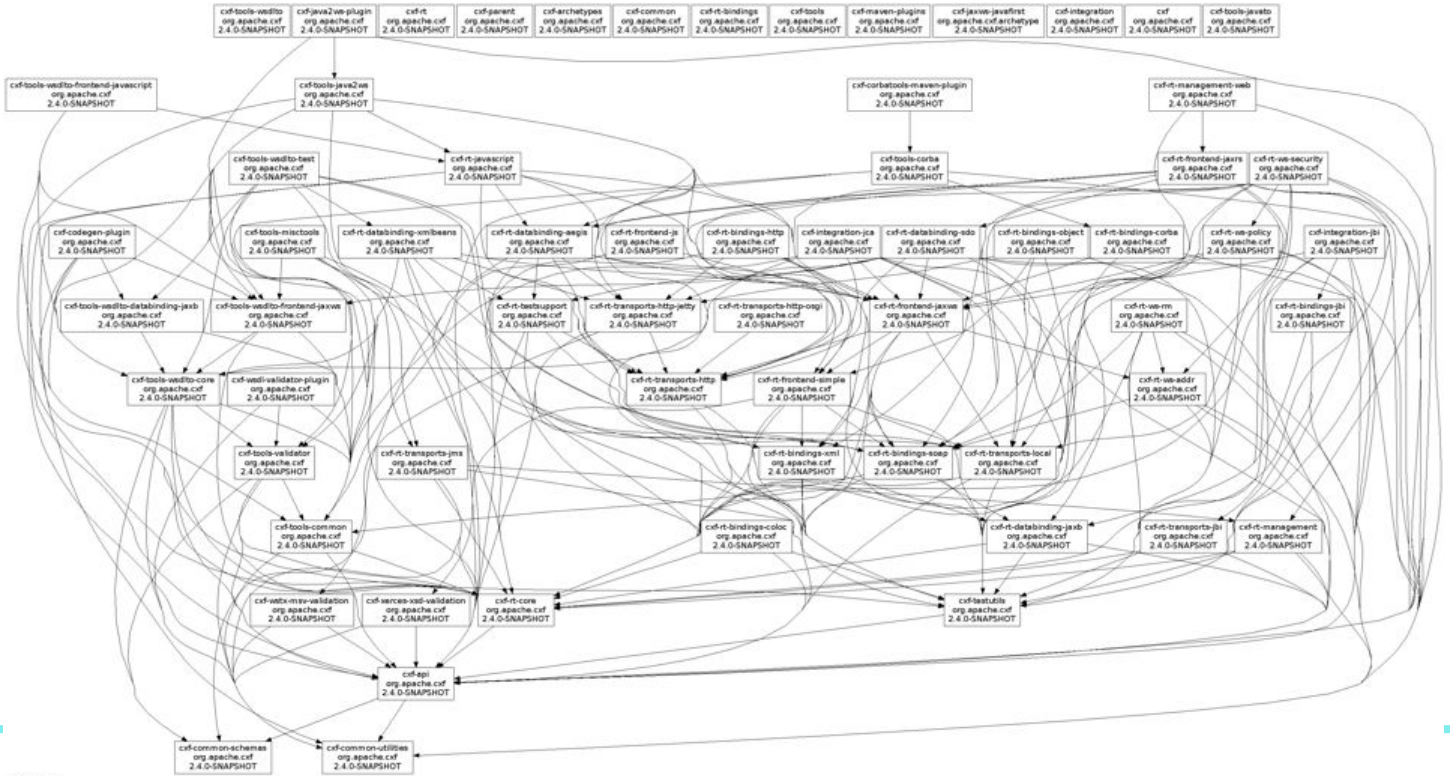
Topological Sort Example #2

What's one valid ordering?

5, 4, 6, 3, 1, 0, 2



Can’t always eyeball it…



Topological Sort Algorithm

- Note that some valid orderings laid out each “child” until we reached a sink.

- We can actually perform a DFS, starting at source nodes and **reverse the DFS finishing times** to get a valid topological sort.

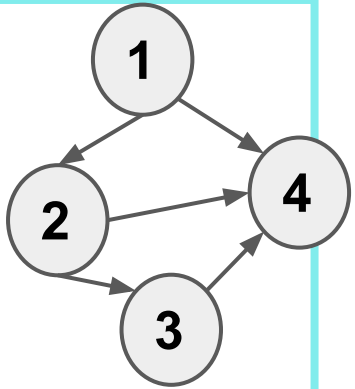
Topological Sort Algorithm

- Note that some valid orderings laid out each “child” until we reached a sink.

- We can actually perform a DFS, starting at source nodes and **reverse the DFS finishing times** to get a valid topological sort.

How do we know which v in G(V, E) are source nodes?

Graph Representation Examples



destination

start

0	**1**	**0**	**1**
0	**1**	**1**	**1**
0	**0**	**0**	**1**
0	**0**	**0**	**0**

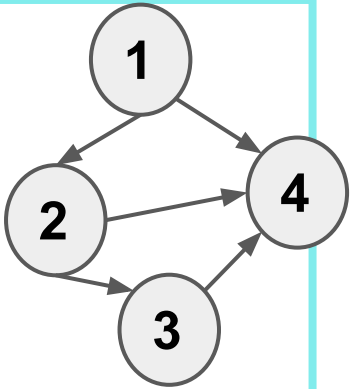
Adjacency Matrix

1: [2, 4]
2: [4, 2, 3]
3: [4]
4: []

Adjacency List

How do we know which v in G(V, E) are source nodes?

Graph Representation Examples



destination

start

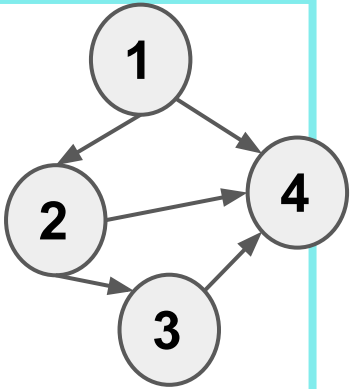
0	1	0	1
0	1	1	1
0	0	0	1
0	0	0	0

Adjacency Matrix

- **Sources**: ???
- **Sinks**: ???

How do we know which v in G(V, E) are source/sink nodes?

Graph Representation Examples



destination

start

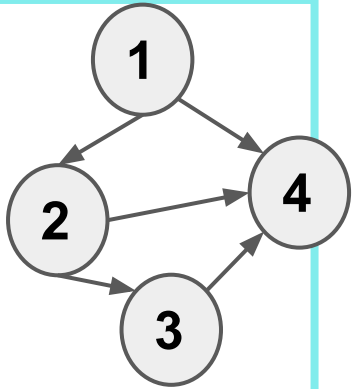
0	1	0	1
0	1	1	1
0	0	0	1
0	0	0	0

Adjacency Matrix

- **Sources**: Columns of all 0s
- **Sinks**: Rows of all 0s

How do we know which v in G(V, E) are source/sink nodes?

Graph Representation Examples



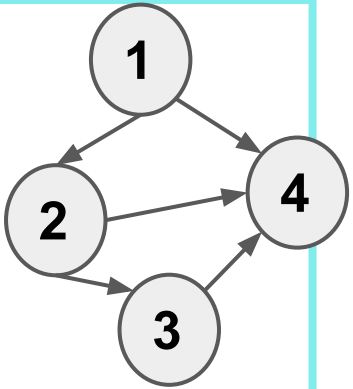
- **Sources**: ???
- **Sinks**: ???

1: [2, 4]
2: [4, 2, 3]
3: [4]
4: []

Adjacency List

How do we know which v in G(V, E) are source nodes?

Graph Representation Examples



- **Sources**: Node that doesn't appear in any list
- **Sinks**: Node that has empty list

1: [2, 4]
2: [4, 2, 3]
3: [4]
4: []

Adjacency List

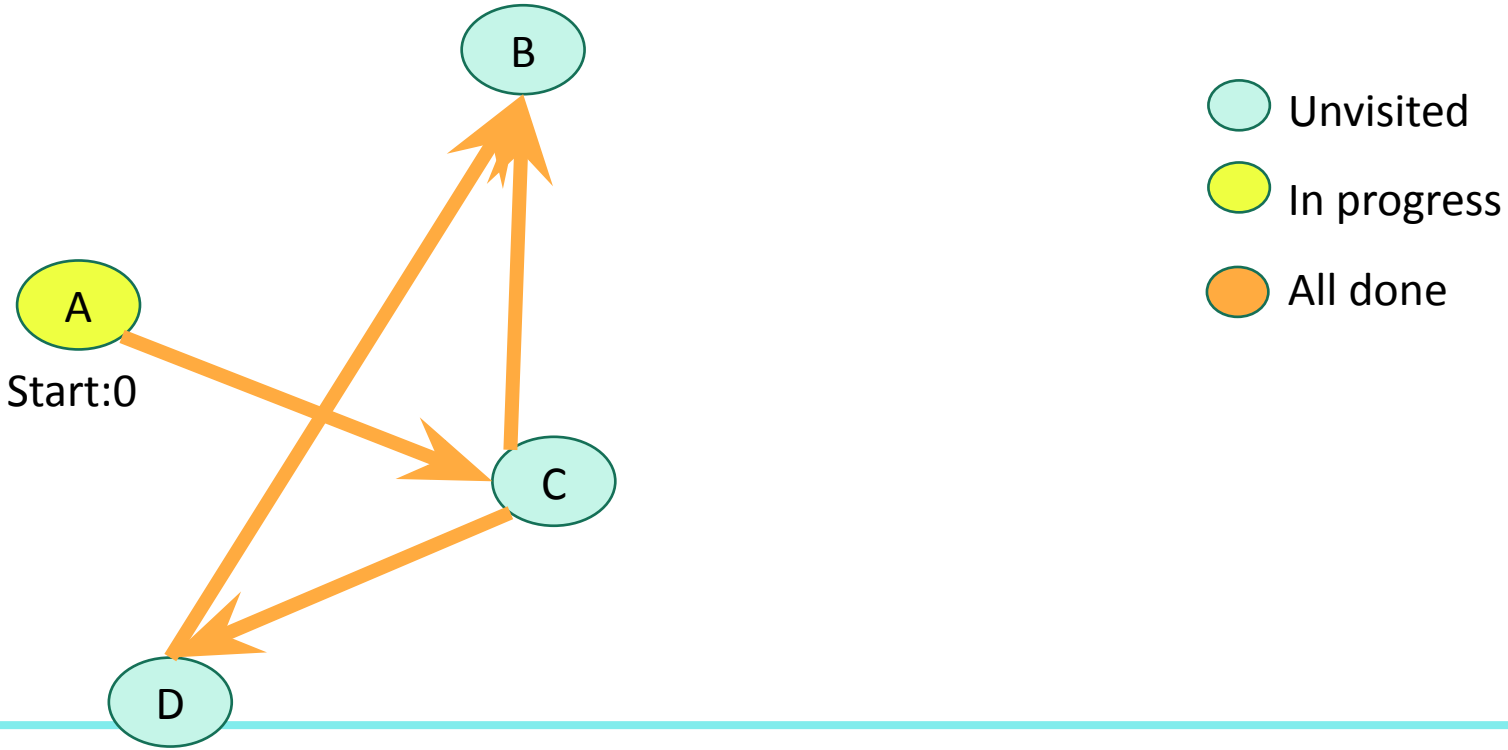
How do we know which v in G(V, E) are source nodes?

Topological Sort Algorithm

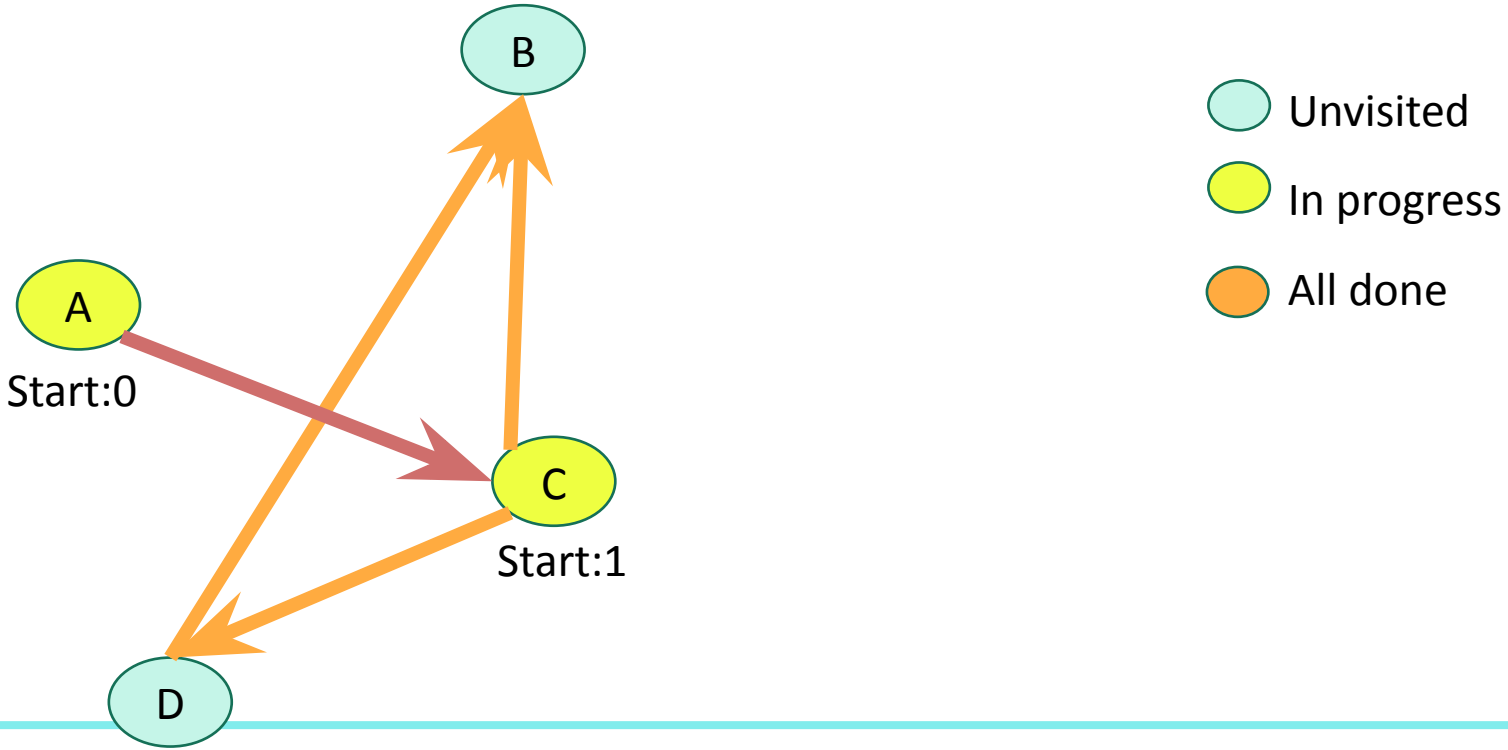
- Note that some valid orderings laid out each “child” until we reached a sink.

- We can actually perform a DFS, starting at source nodes and **reverse the DFS finishing times** to get a valid topological sort.

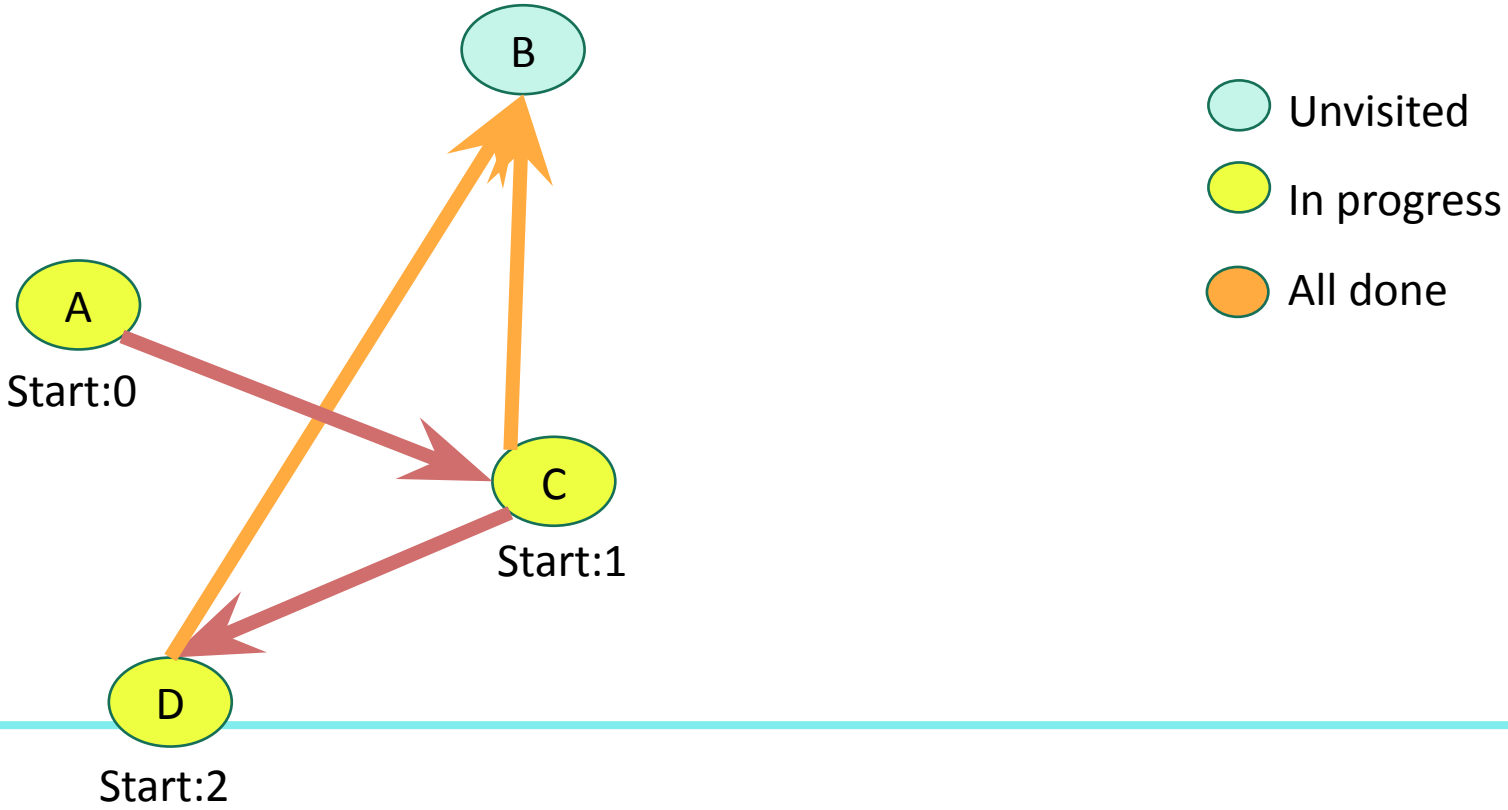
TopoSort Example



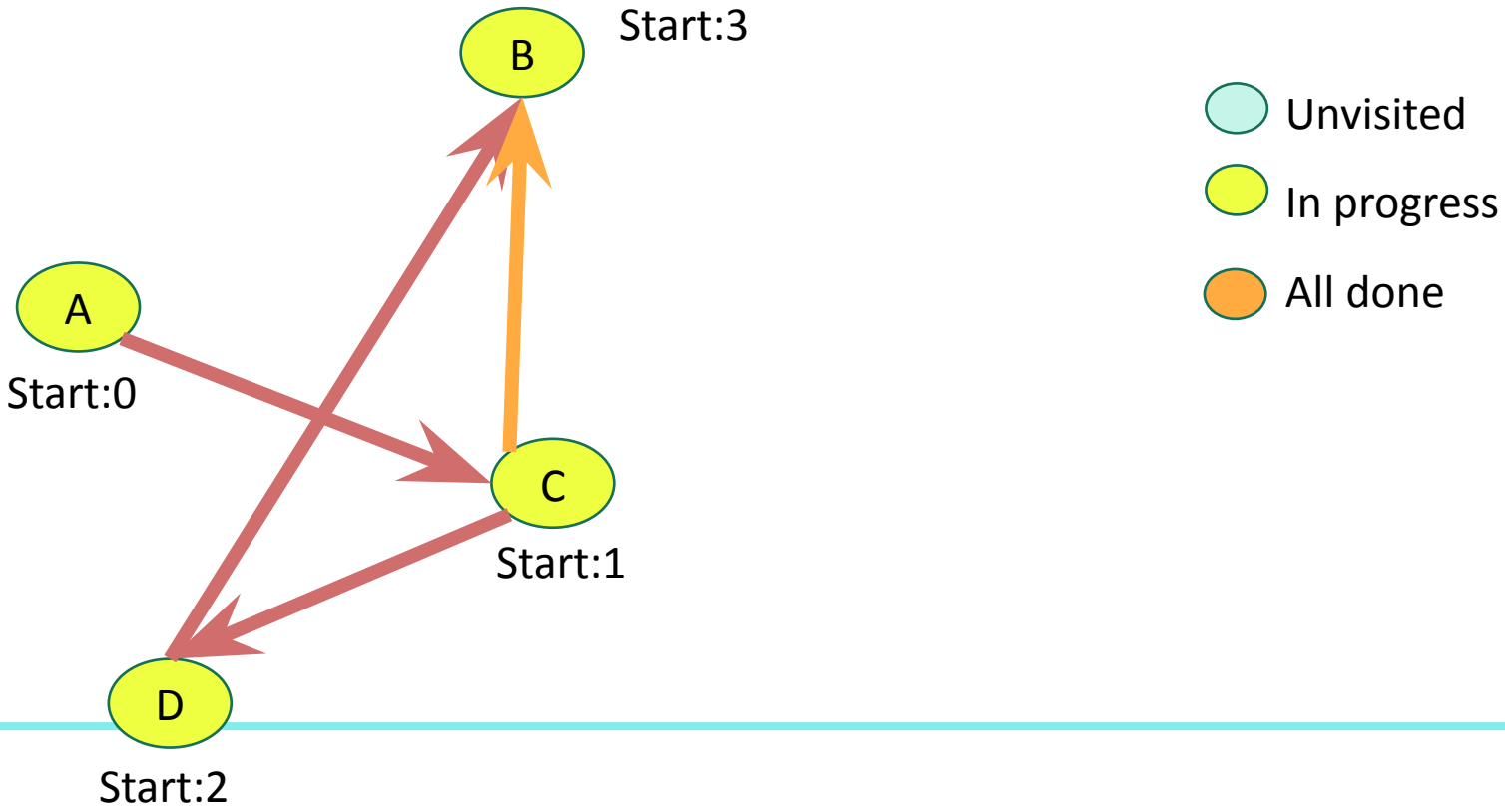
TopoSort Example



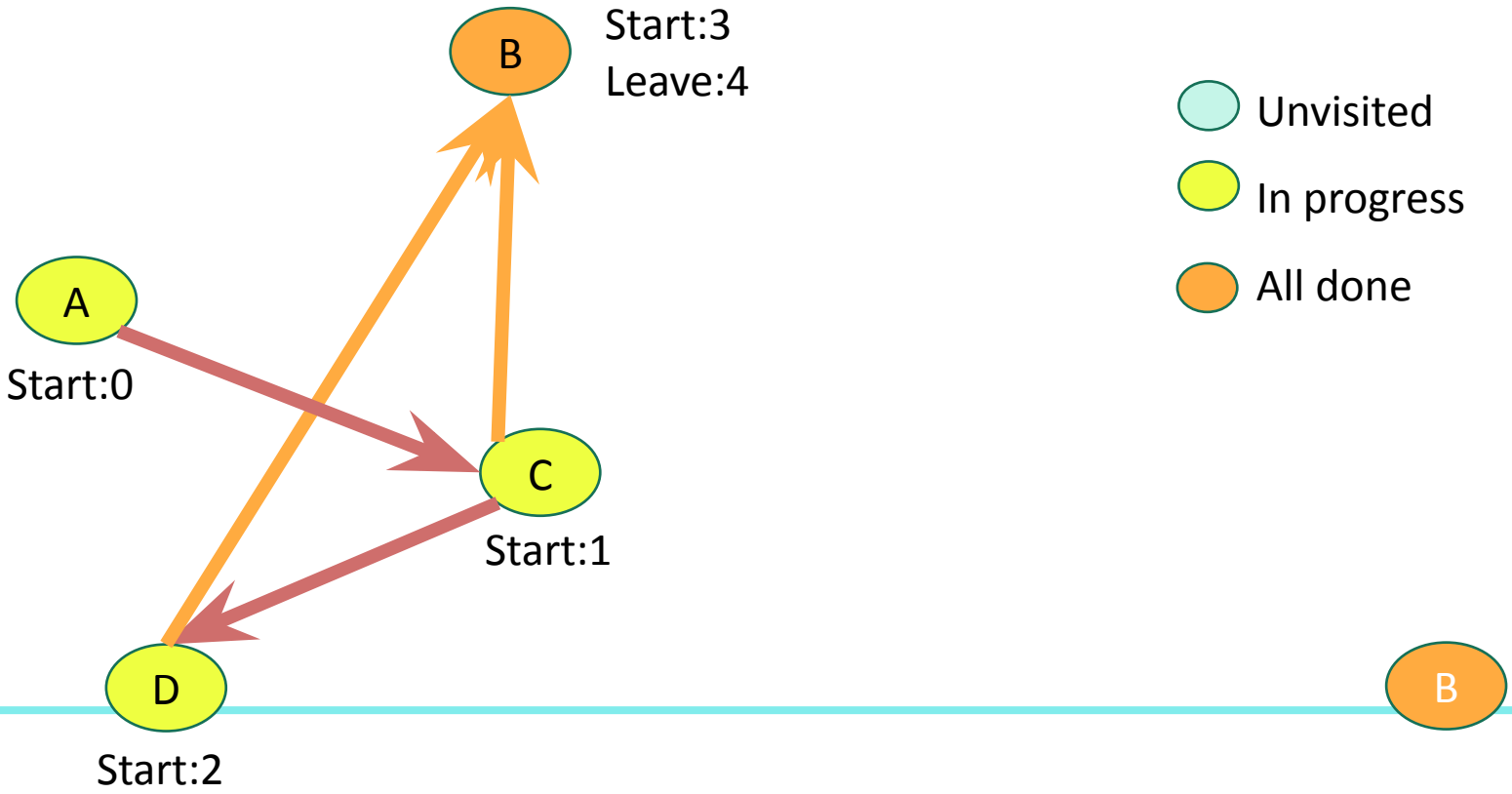
TopoSort Example



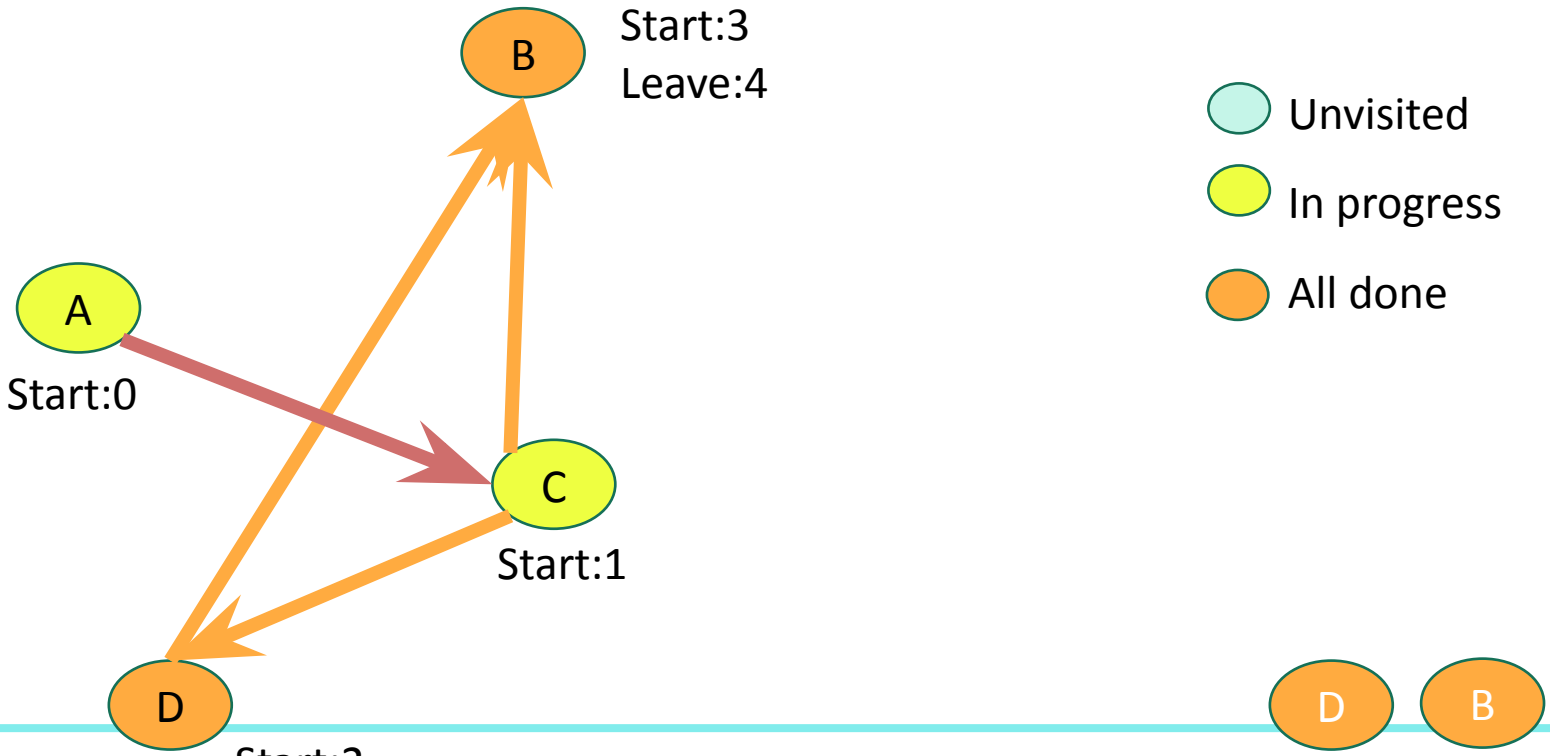
TopoSort Example



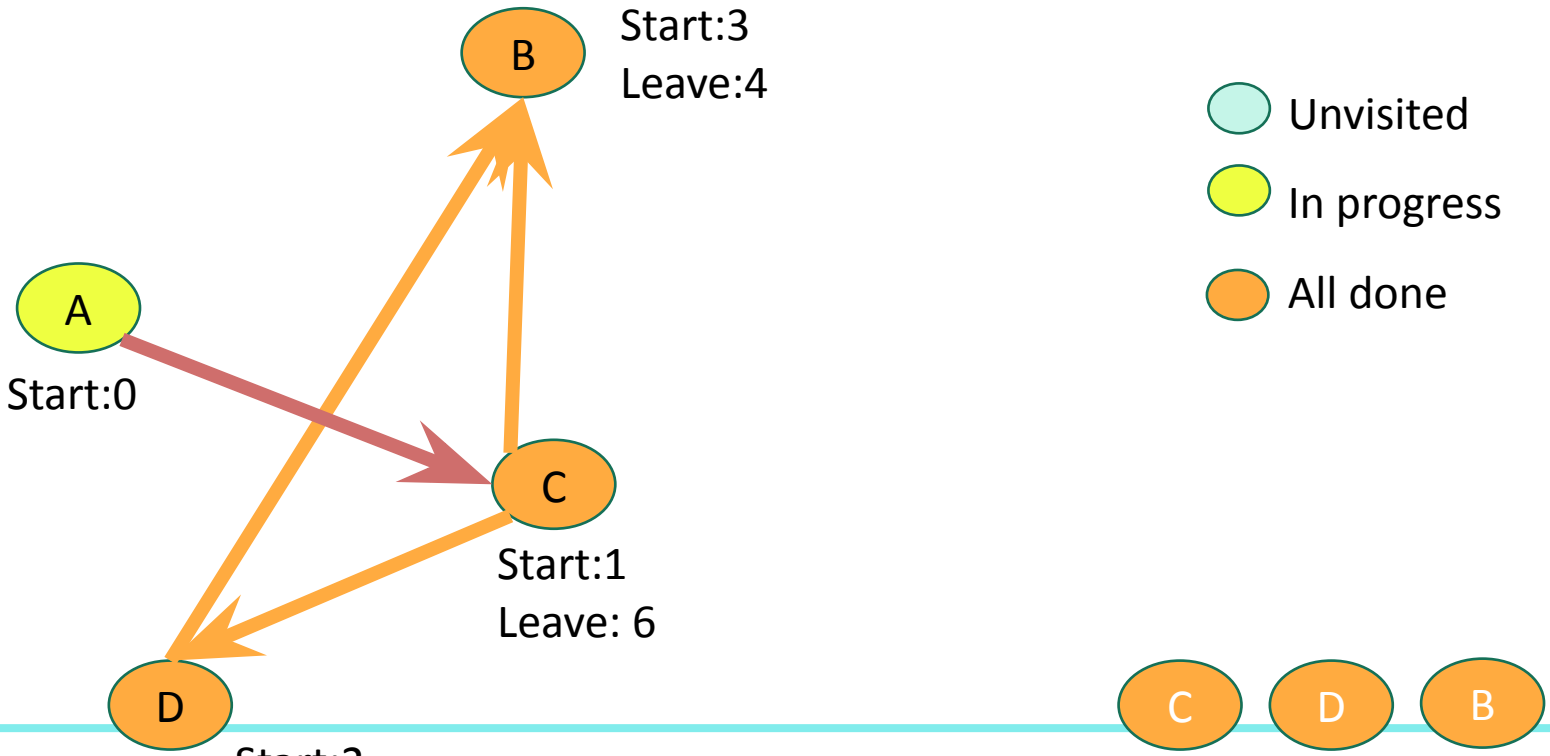
TopoSort Example



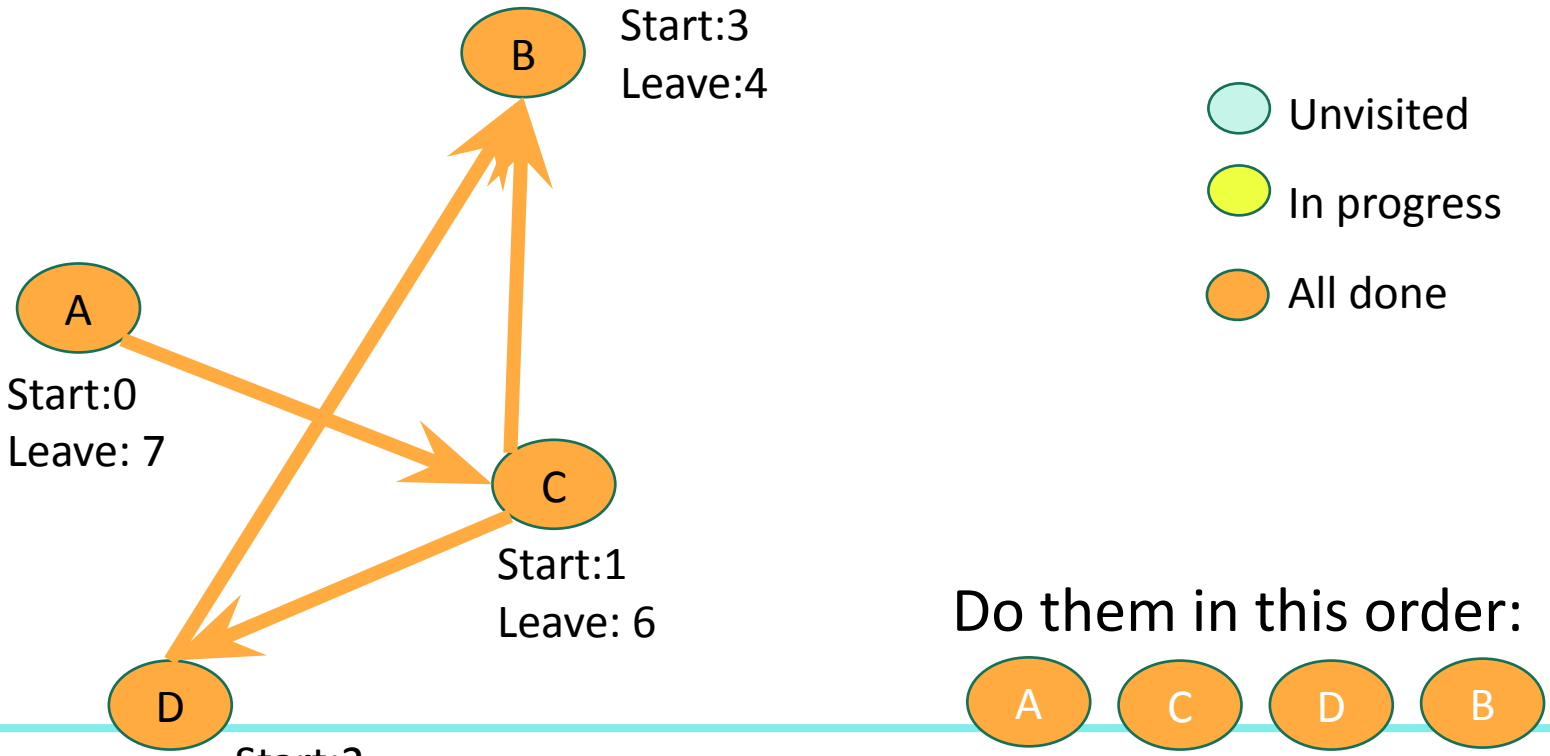
TopoSort Example



TopoSort Example



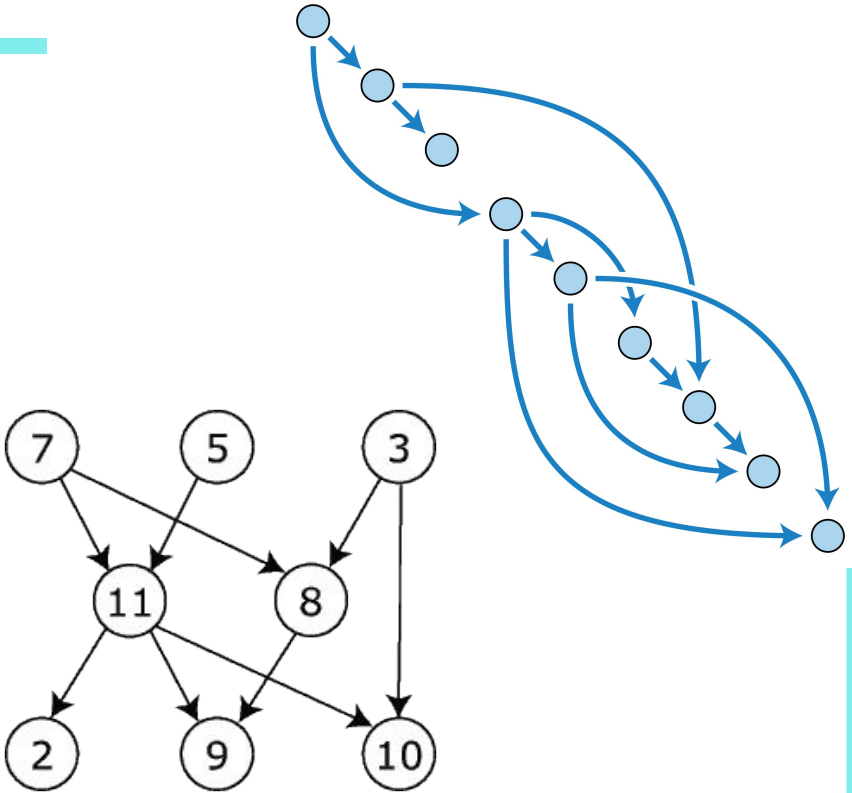
TopoSort Example



TopoSort Code

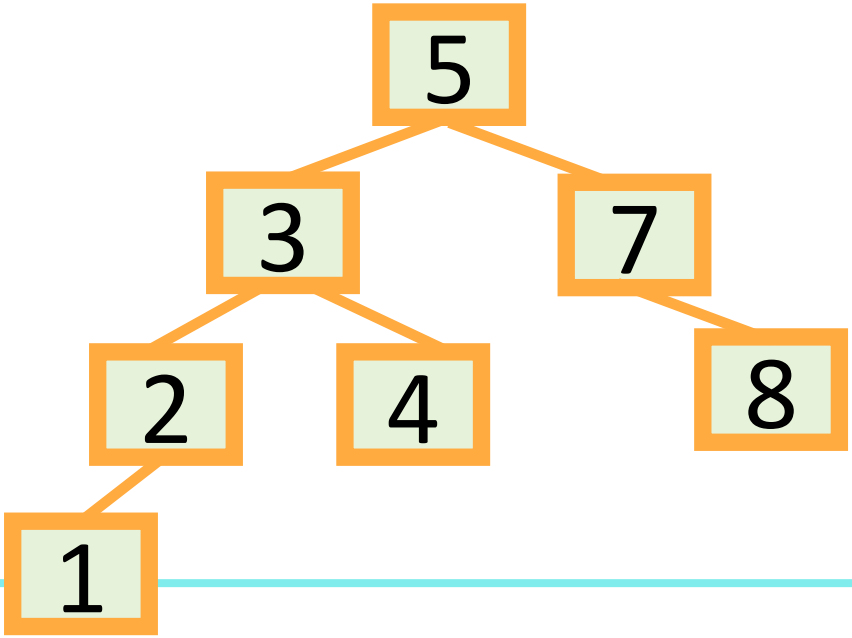
For implementation,

see repl.it



Another use of DFS that we've already seen…

- In-order enumeration of binary search trees



Perform DFS and print a node's label when you are done with the left child and before you begin the right child.

COMP - 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 15: Graph DFS + Topological Sort

Lecturer: Chris Lucas (cflucas@ncat.edu)