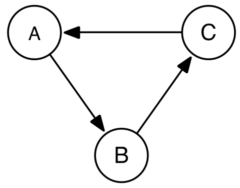
(08) Graphs Part 2: Topological Sorting

Video (12 mins): https://youtu.be/JfRAzuqyHZ0

Precedence or Dependency Relationship

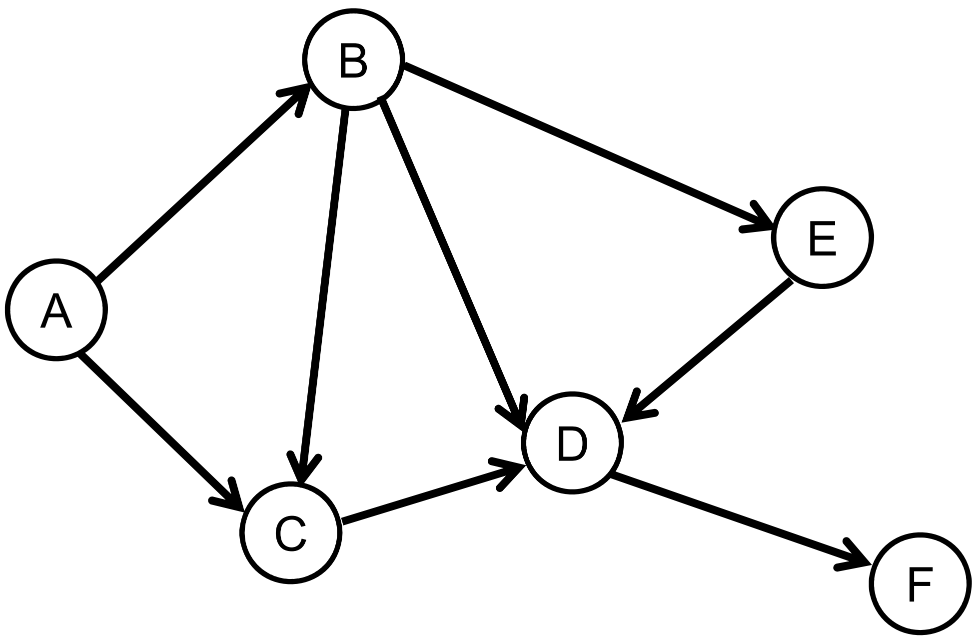
- Directed graph can be used to model dependency relationship
  - directed edges indicate the direction of dependency
  - A-to-B edge means B depends on A

- Examples of dependency relationships:
  - Course prerequisites
  - Task scheduling
  - Causal or temporal relationships



- Cycles in such settings might be problematic:
  - Say IS200 is prerequisite for IS201, and IS201 is prerequisite for IS103, and IS103 is prerequisite for IS200.

Directed “Acyclic” Graph (DAG)



Adjacency List

A	B	C	
B	C	D	E
C	D		
D	F		
E	D		
F			

A DAG does not contain any cycle

Topological Ordering

- A sequence or permutation of all vertices in a graph G, such that all edges "point forward".
- For every edge in G, the source vertex appears before the target vertex in a topological ordering.
- Any DAG always contains at least one topological ordering.
- Some DAGs contain more than one.
- Application:
  - Finding a course schedule that satisfies the prerequisite requirements.
  - Scheduling tasks such that when a task is ready to execute, all the tasks it depends on have already completed.
  - Finding a chain of cause and effect that may lead to a particular event.

Example: Topological Ordering

Figure 14-14

A DAG

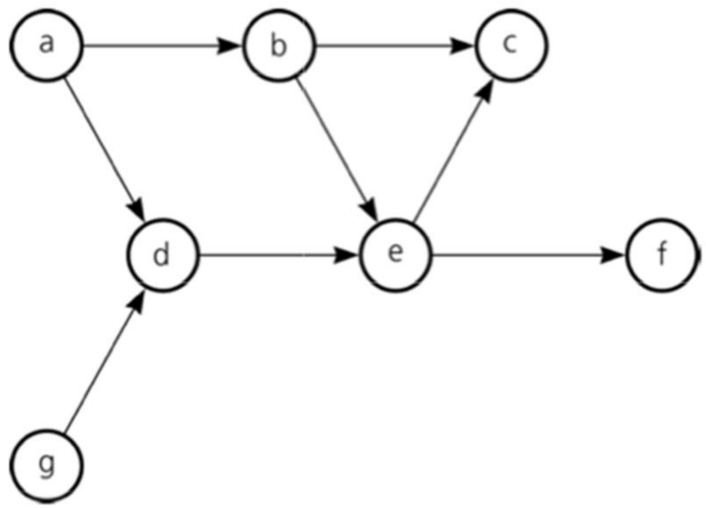
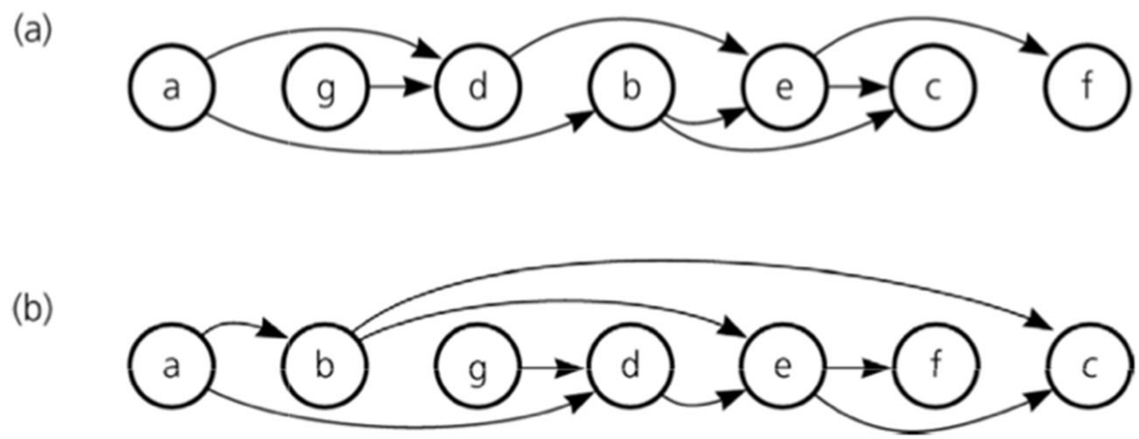


Figure 14-15

The graph in Figure 14-14 arranged according to the topological orders:

a) *a, g, d, b, e, c, f* and

b) *a, b, g, d, e, f, c*



Prichard and Carrano, "Data Abstraction & Problem Solving with Java", 3rd edition, Pearson.

Topological Sort

- How to find a topological ordering of vertices in a DAG?
- For any path in the graph, the vertices will appear in the same order as in any valid topological ordering.
  - There may be some other interleaving vertices.
- The traversal that explores the graph by following paths is DFS.
- In DFS traversal, by the time the recursive call on the current vertex `v` is completed, any vertex reachable from `v` through some path will already be visited.
  - All other reachable vertices should follow `v` in a topological ordering.
- Strategy: add `v` into a stack as the recursive call completes.

Topological Sort Algorithm

```
def topsort(graph):
  s = Stack()
  for i in range(len(graph.vertices)):
    vi = graph.vertices[i]
    if not vi.isVisited():
      topsort_dfs(vi, s)
  return s
```

Run DFS from every vertex to ensure that all vertices are included in a topological ordering.

Topological Sort Algorithm

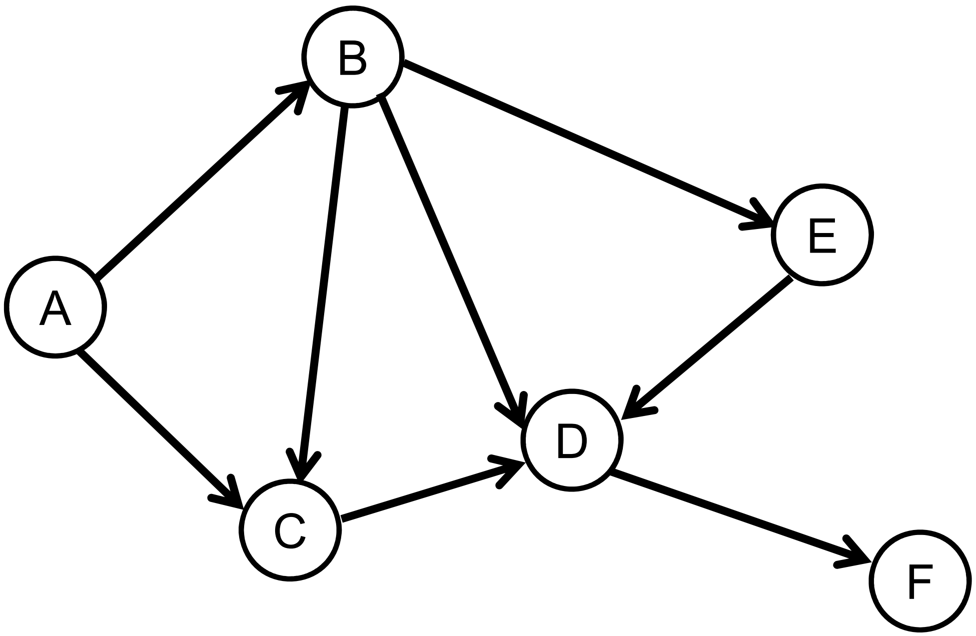
```
def topsort_dfs(vertex, stack):
  visit(vertex)
  for i in range(len(vertex.adjList)):
    neighbor = vertex.adjList[i]
    if not neighbor.isVisited():
      topsort_dfs(neighbor, stack)
  stack.push(vertex)
```

After all my neighbours have been visited, I push myself onto stack

Add vertex into a stack at the end of DFS traversal on this vertex.

- Vertices at the end of a path will be pushed first into the stack.
- When a vertex is pushed into the stack, all vertices depending on it are already in the stack.

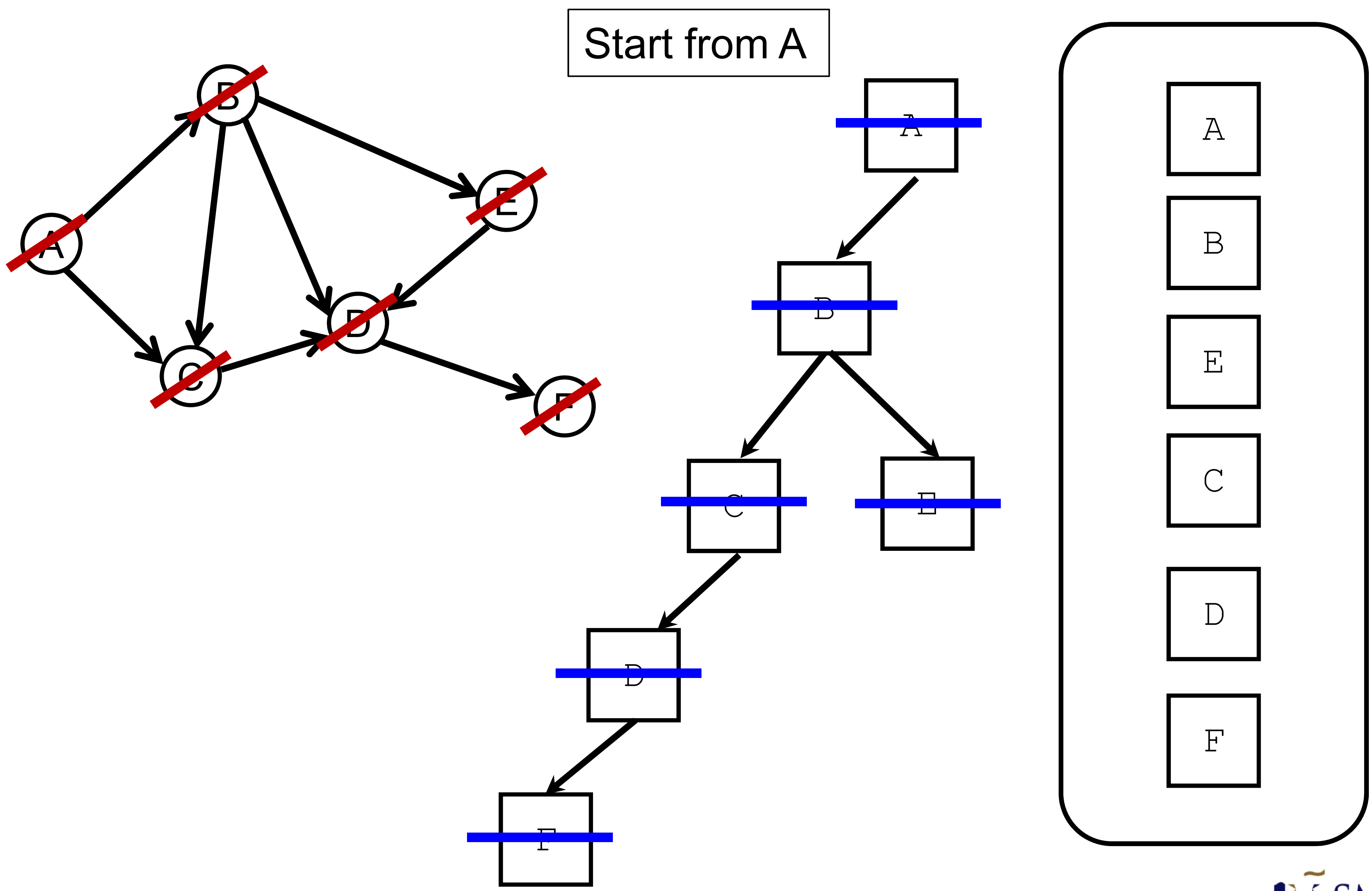
Example: Topological Sort



Adjacency List

A	B	C	
B	C	D	E
C	D		
D	F		
E	D		
F			

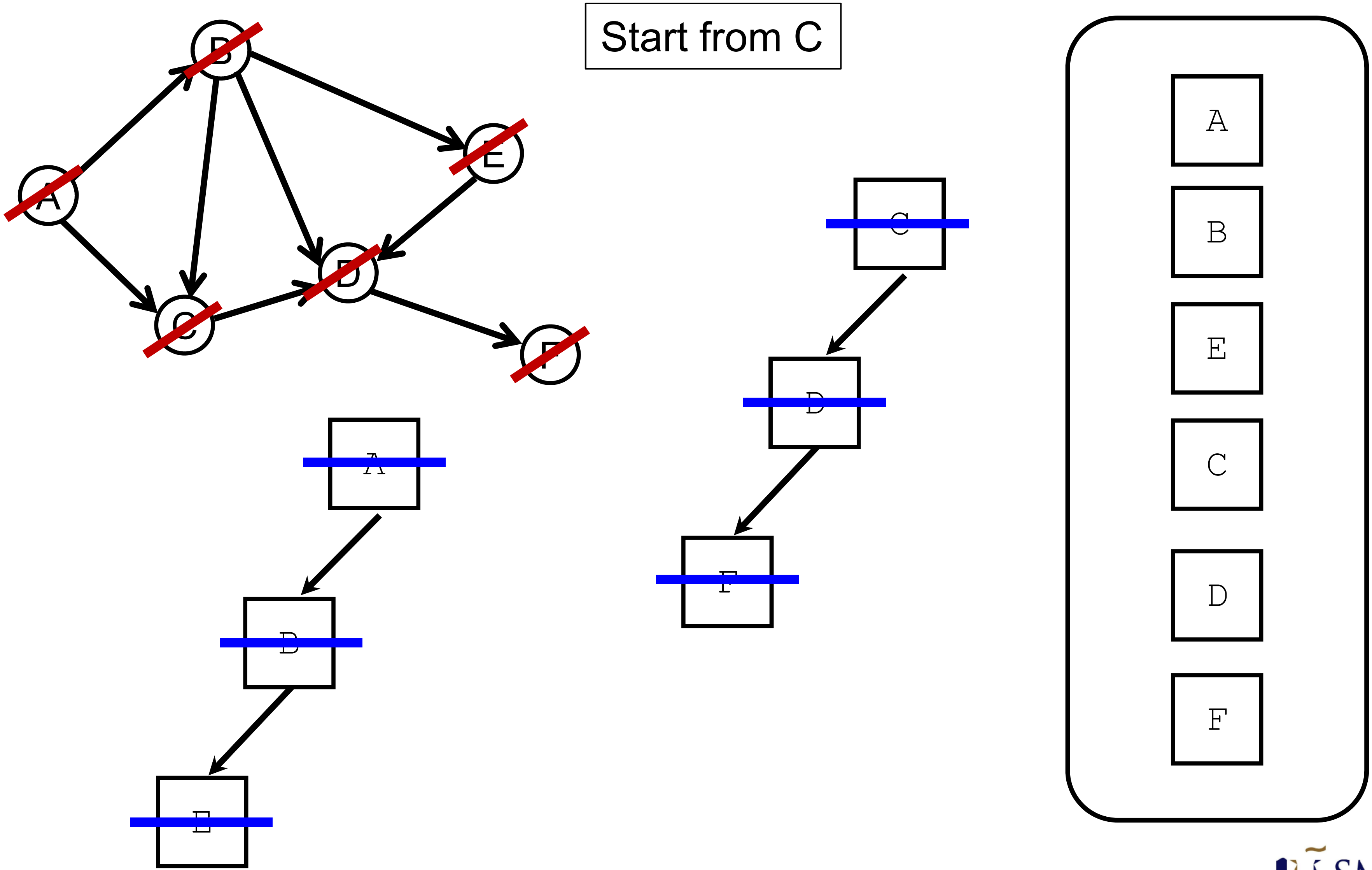
Example: Topological Sort



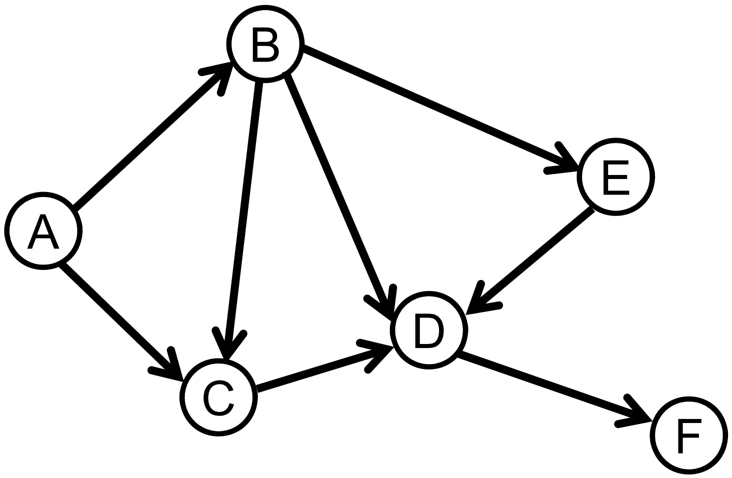
Example: Topological Sort

TO Stack

Start from C

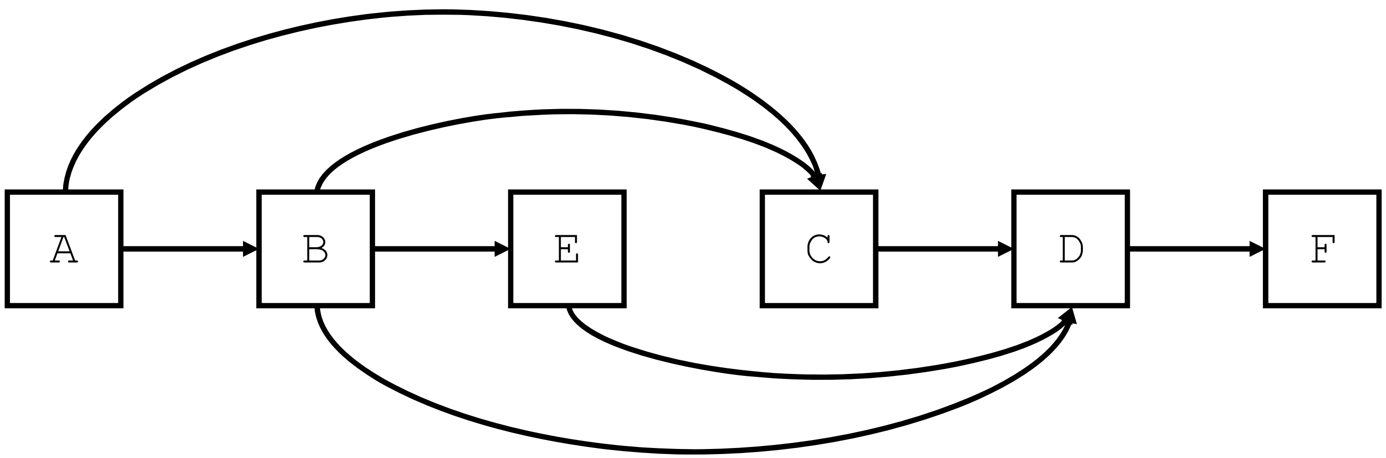


Confirming the Topological Order



Adjacency List

A	B	C	
B	C	D	E
C	D		
D	F		
E	D		
F			



Practice Topological Sort

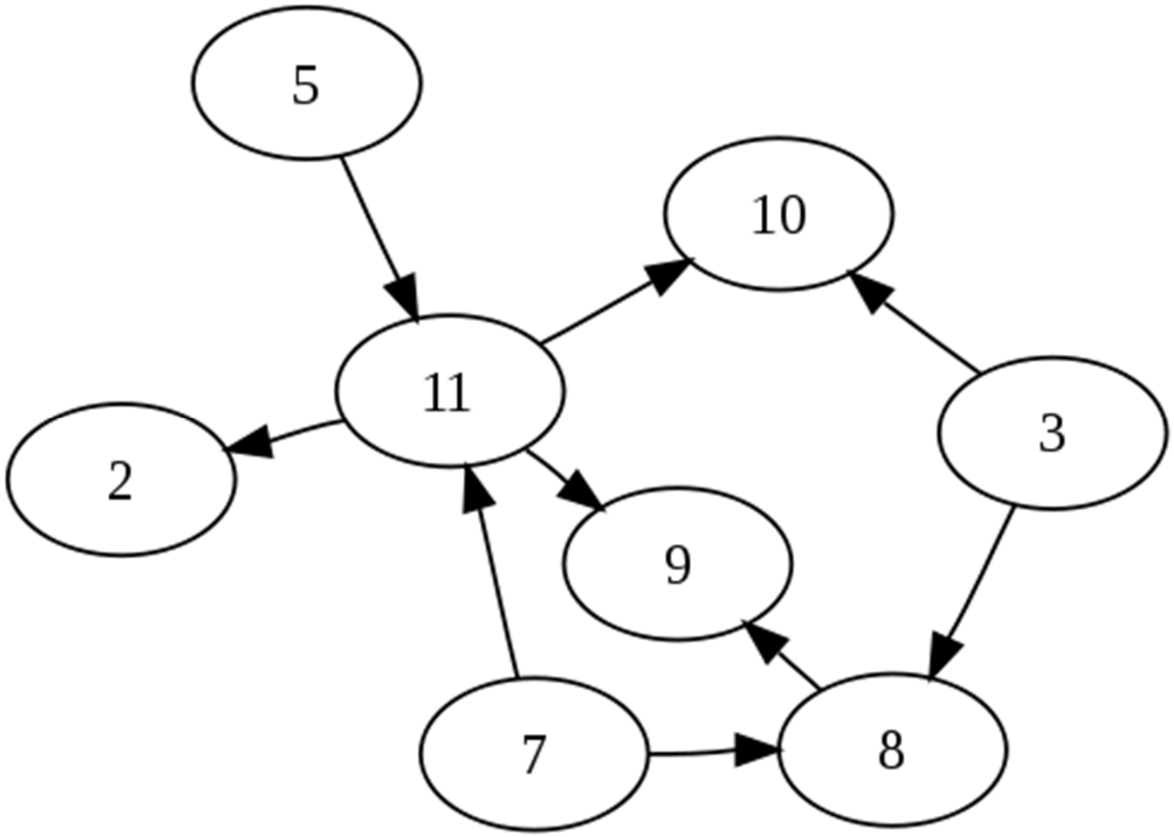


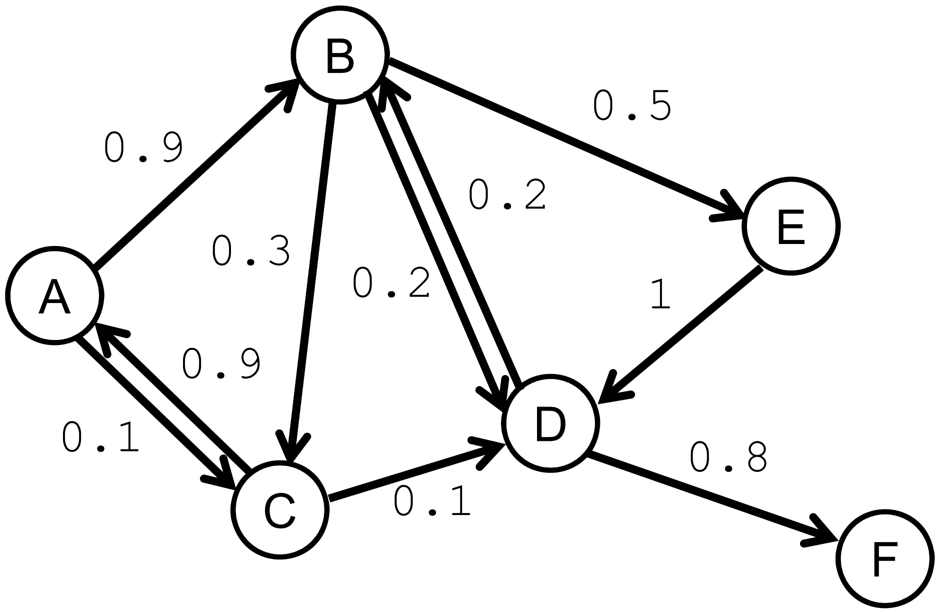
Diagram source: http://en.wikipedia.org/w/index.php?title=File:Directed_acyclic_graph_3.svg

Weighted Edges

- Binary value (0 or 1) only allows us to model existence of edges.
  - Path length measured in terms of number of edges.

- Continuous values are more flexible to model varying "importance" of edges.
  - In a road network, edges may have other information such as the distance travelled, the time taken, the cost.
  - Path length measured in terms of aggregating the values (e.g., summation):
    - Total distance travelled.
    - Total time taken.
    - Total cost.

Weighted Graph



Adjacency List

A	(B, 0.9)	(C, 0.1)	
B	(C, 0.3)	(D, 0.2)	(E, 0.5)
C	(D, 0.1)	(A, 0.9)	
D	(B, 0.2)	(F, 0.8)	
E	(D, 1)		
F			

Adjacency Matrix

	A	B	C	D	E	F
A	∞	0.9	0.1	∞	∞	∞
B	∞	∞	0.3	0.2	0.5	∞
C	0.9	∞	∞	0.1	∞	∞
D	∞	0.2	∞	∞	∞	0.8
E	∞	∞	∞	1	∞	∞
F	∞	∞	∞	∞	∞	∞

Shortest Path

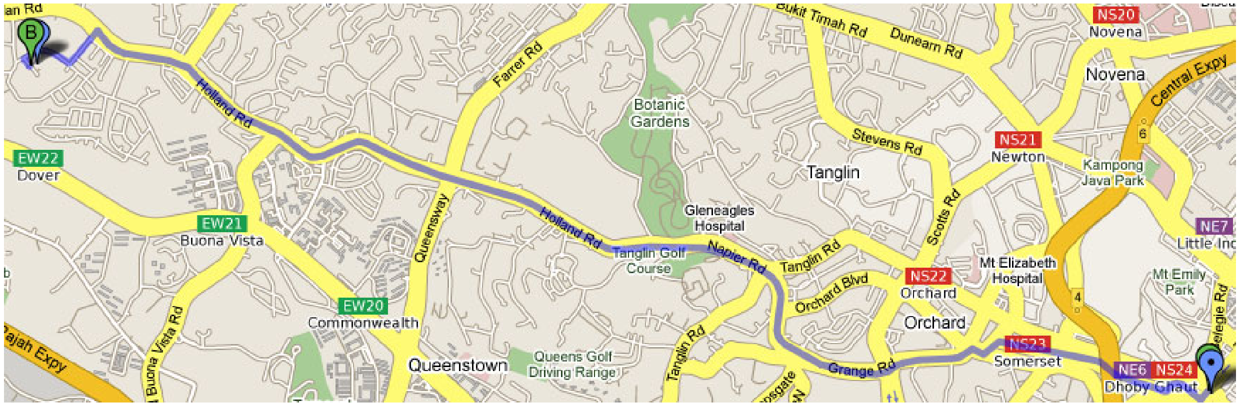
- In a weighted graph, assuming non-negative weights of edges.
- Find the shortest path from a vertex `v1` to another vertex `v2`.
  - "Path length" is a sum of the edge weights in the path.
- In an unweighted graph, the shortest path has fewest number of edges.
- In a weighted graph, the shortest path may not have the fewest edges, but have the lowest aggregate weight.
  - (A, B) has length 1, but weight of 0.9
  - (A, C, D, B) has length 3, but weight of 0.4

shortest_path(graph, start, end)

Returns a tuple in the format of (distance, sequence)

Example: Routing

- Finding shortest (or least-cost) path:



- Very useful in logistic planning, transportation, network communication, etc.
- Any problem with state space graph can be solved by routing (shortest path from initial state to goal).

Summary

- Graphs are data structures that can represent network relationships
- Edge direction:
  - Directed graph: a graph may have directed edges
  - Undirected graph: a graph with undirected (bidirectional) edges
- Edge weights:
  - Binary: edges are either present or absent
  - Weighted: edges can have continuous values as weights
- Traversal:
  - depth-first search (DFS)
  - breadth-first search (BFS)
- Algorithm:
  - Topological sorting to find topological ordering in a DAG

Road Map

Algorithm Design and Analysis

(Weeks 1 - 5)

Fundamental Data Structures

(Weeks 6 - 9)

Computational Intractability and Heuristic Reasoning

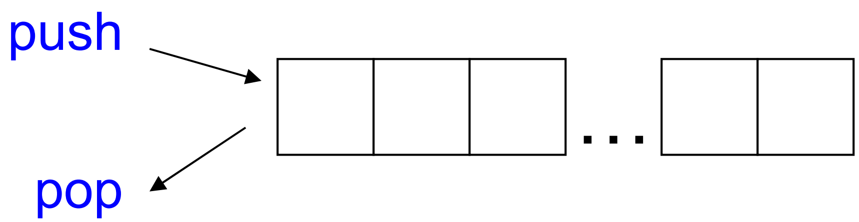
Next week ⟶

- Week 10: Heuristics
- Week 11: Limits of Computation
- Week 13: Review

The following slides (DFS with stacks) are not covered in the video, and will be covered in class.

DFS (with Stack)

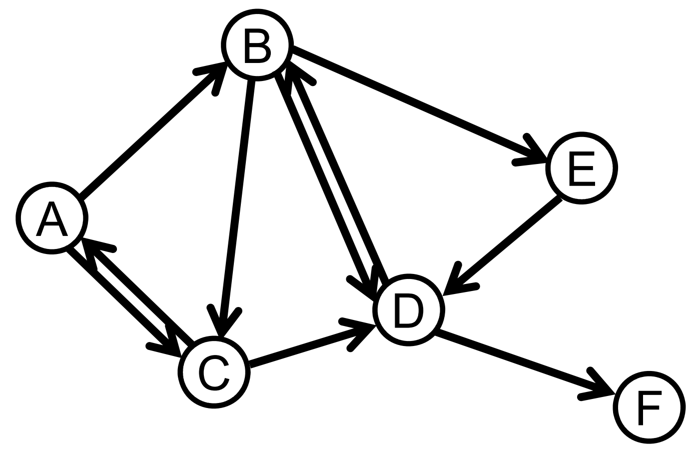
- List of candidate nodes stored in a **stack**: last-in-first-out (LIFO).
  - Get operation returns the **newest** item.



- Always expand the **deepest** node.

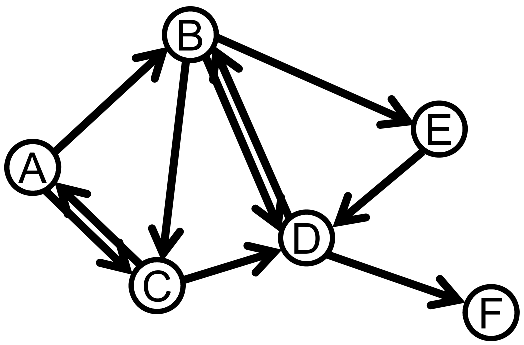
DFS (with Stack)

```
def dfs(vertex):
  s = Stack()
  s.push(vertex)
  while s.count() > 0:
    v = s.pop()
    if not v.isVisited():
      visit(v)
    for i in range(len(v.adjList)):
      n = v.adjList[len(v.adjList) - 1 - i]
      if not n.isVisited():
        s.push(n)
```



To pop according to alphabetic order.

Example: DFS (with Stack)

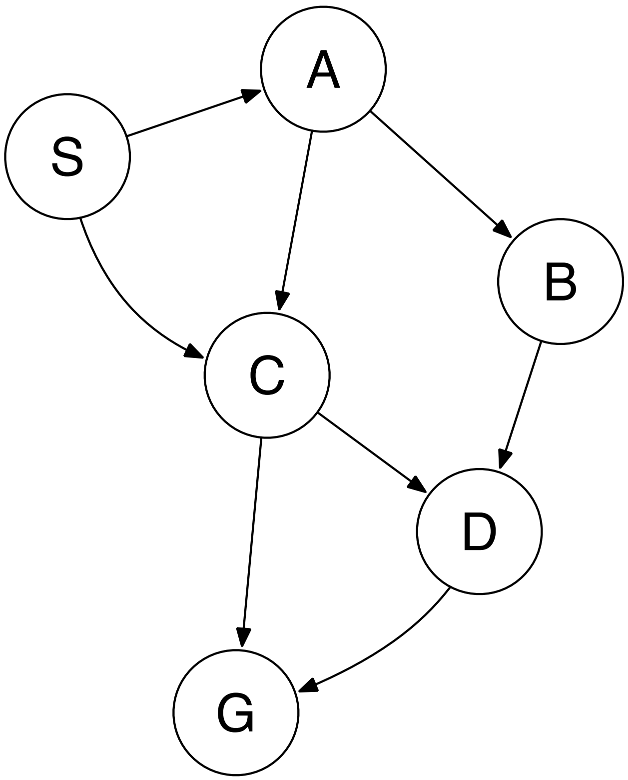


Adjacency List

A	B	C	
B	C	D	E
C	D	A	
D	B	F	
E	D		
F			

step	popped	visited	pushed	stack: [B..T]
0			A	[A]
1	A	A	C B	[C B]
2	B	B	E D C	[C E D C]
3	C	C	D	[C E D D]
4	D	D	F	[C E D F]
5	F	F		[C E D]
6	D			[C E]
7	E	E		[C]
8	C			

Practice DFS (with Stack)



step	popped	visited	pushed	stack
0			S	[S]
1				
2				
3				
4				
5				
6				
7				
8				

Complexity of DFS/BFS

https://www.quora.com/Why-is-the-complexity-of-DFS-O-V+E

Quora Home 1 Answer 1 Spaces Notifications 8 Search

Originally Answered: Why is the complexity of DFS O(V+E) ?

Say, you have a connected graph with V nodes and E edges.

In DFS, you traverse each node exactly once. Therefore, the time complexity of DFS is at least **O(V)**.

Now, any additional complexity comes from how you discover all the outgoing paths or edges for each node which, in turn, is dependent on the way your graph is implemented. If an edge leads you to a node that has already been traversed, you skip it and check the next. Typical DFS implementations use a hash table to maintain the list of traversed nodes so that you could find out if a node has been encountered before in O(1) time (constant time).

- If your graph is implemented as an adjacency matrix (a V x V array), then, for each node, you have to traverse an entire row of length V in the matrix to discover all its outgoing edges. Please note that each row in an adjacency matrix corresponds to a node in the graph, and the said row stores information about edges stemming from the node. So, the complexity of DFS is **O(V * V) = O(V^2)**.
- If your graph is implemented using adjacency lists, wherein each node maintains a list of all its adjacent edges, then, for each node, you could discover all its neighbors by traversing its adjacency list just once in linear time. For a directed graph, the sum of the sizes of the adjacency lists of all the nodes is E (total number of edges). So, the complexity of DFS is **O(V) + O(E) = O(V + E)**.
    - For an undirected graph, each edge will appear twice. Once in the adjacency list of either end of the edge. So, the overall complexity will be **O(V) + O (2E) ~ O(V + E)**.

Source: https://www.quora.com/Why-is-the-complexity-of-DFS-O-V+E