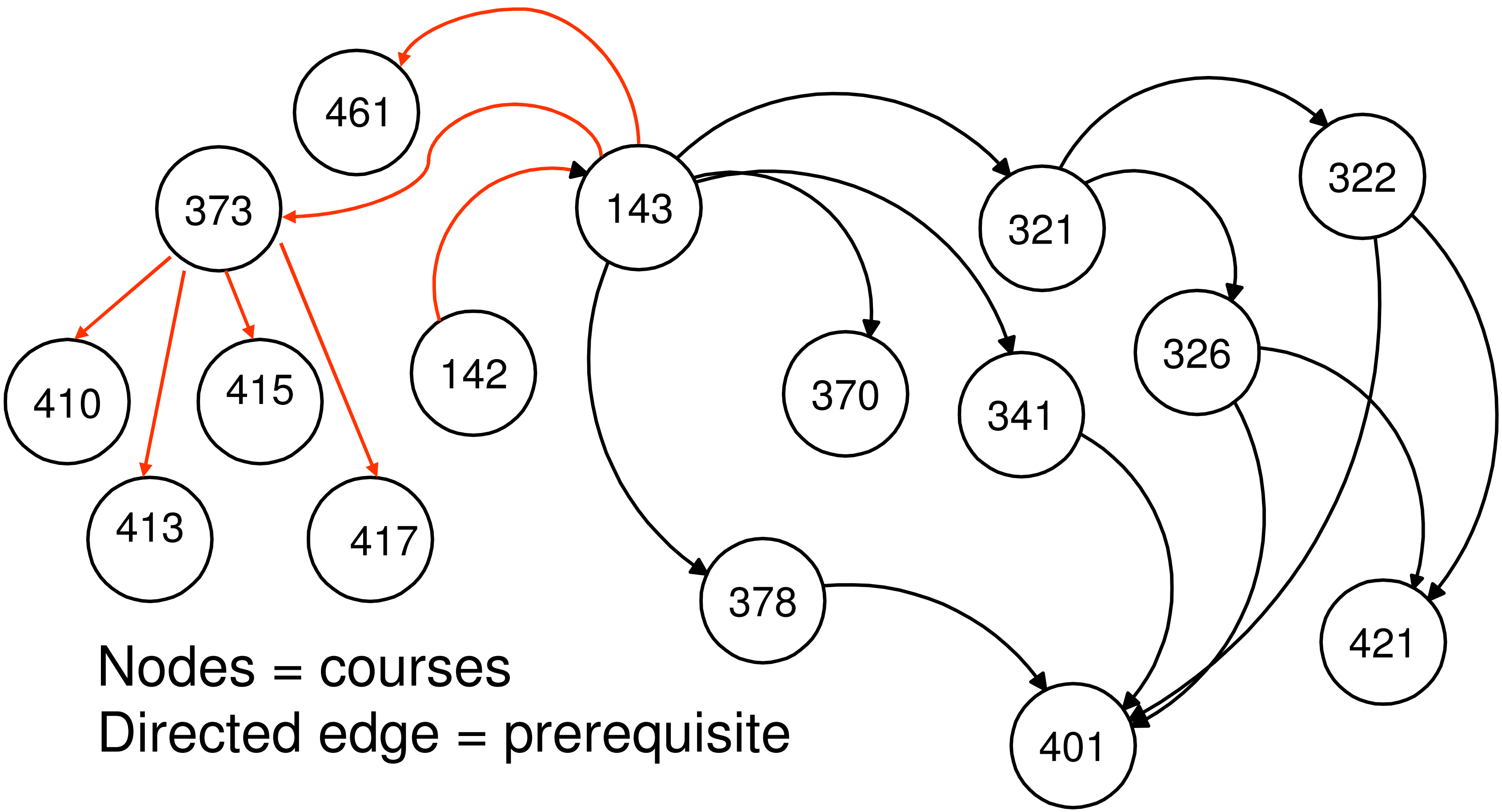
CSE 390B: Graph Algorithms

Based on CSE 373 slides
by Jessica Miller, Ruth Anderson

A Graph: Course Prerequisites

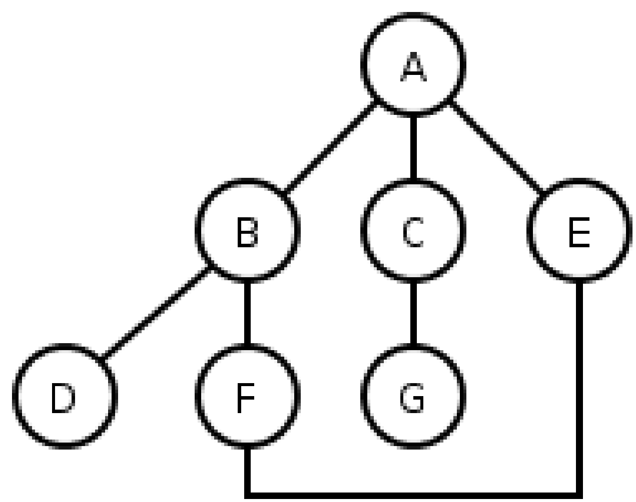


Nodes = courses
Directed edge = prerequisite

Depth-First Search (DFS)

- **depth-first search (DFS)**: find path between two vertices by exploring each path as many steps as possible before backtracking
  - often implemented **recursively** with a **stack** for the path in progress
  - always finds a path, but not necessarily the shortest one
  - easy to reconstruct the path once you have found it (just unroll the calls)

- DFS path search order from A to others (assumes ABC edge order):
  - A
  - A → B
  - A → B → D
  - A → B → F
  - A → B → F → E
  - A → C
  - A → C → G



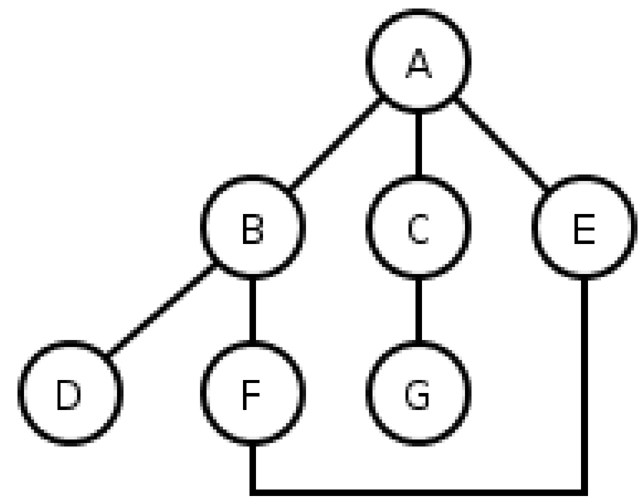
DFS pseudocode

dfs(v1, v2):
  path = new Stack().
  dfs(v1, v2, path)

dfs(v1, v2, path):
  *path.**push**(v1).*
  ***mark** v1 as visited.*
  if v1 = v2:
    path is found.

  for each unvisited neighbor v_i of v1 with edge (v1 $\rightarrow$ v_i):
    if dfs(v_i, v2, path) finds a path, path is found.

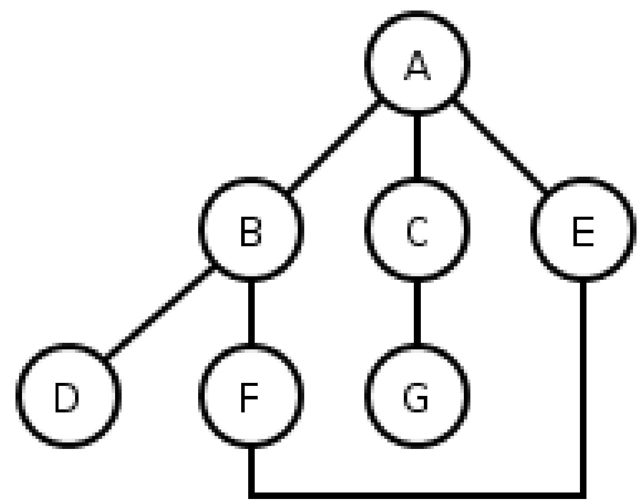
  *path.**pop**().*   *// path is not found.*



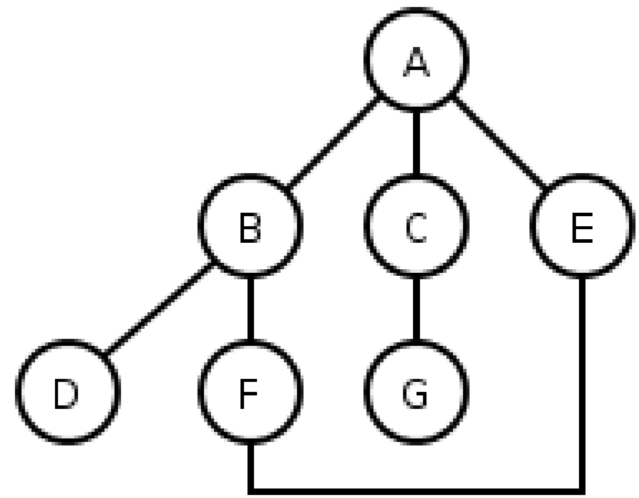
Breadth-First Search (BFS)

- **breadth-first search (DFS)**: find path between two nodes by taking one step down all paths and then immediately backtracking
  - often implemented with a **queue** of next vertices to visit
  - always finds the **shortest path** (fewest edges); optimal for unweighted graphs
  - harder to reconstruct the path once you have found it

- BFS path search order from A to others:
  - A
  - A → B
  - A → C
  - A → E
  - A → B → D
  - A → B → F
  - A → C → G



BFS pseudocode

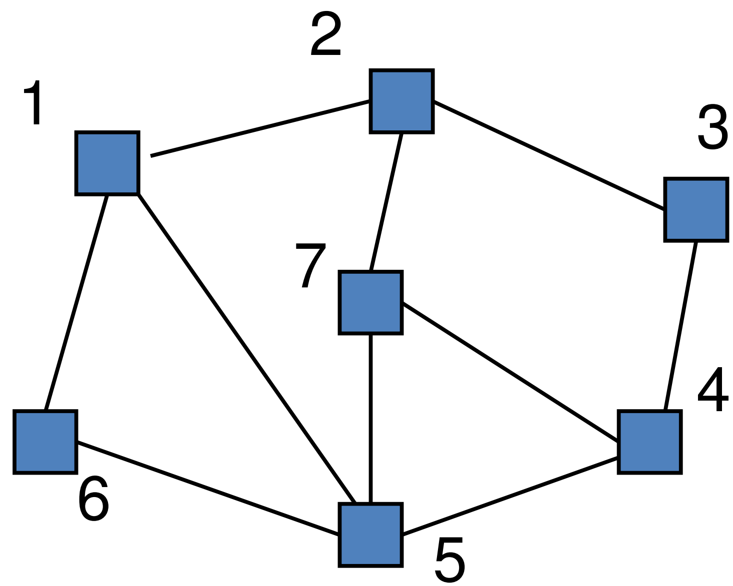



```
bfs(v1, v2):
  Q = {v1}.
  mark v1 as visited.

  while Q not empty:
    v = Q.dequeue().      // remove from front
    if v is v2:
      path is found.

  for each unvisited neighbor vi of v1 with edge (v1 → vi):
     mark vi as visited.
     Q.enqueue(vi).      // add at end

  path is not found.
```

Implementation: Adjacency Matrix

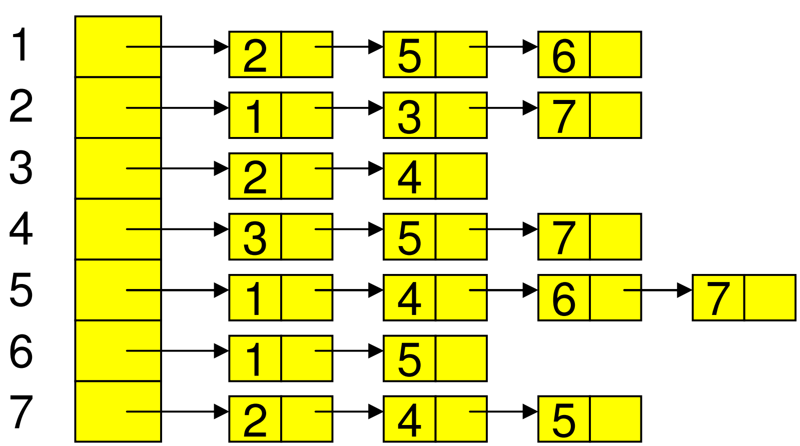
- an $n \times n$ 2D array where M[a][b] = edges from v_a to v_b
  - Sometimes implemented as a Map<V, Map<V, E>>
  - Good for quickly asking, "is there an edge from vertex *i* to *j*?"
  - How do we figure out the degree of a vertex?

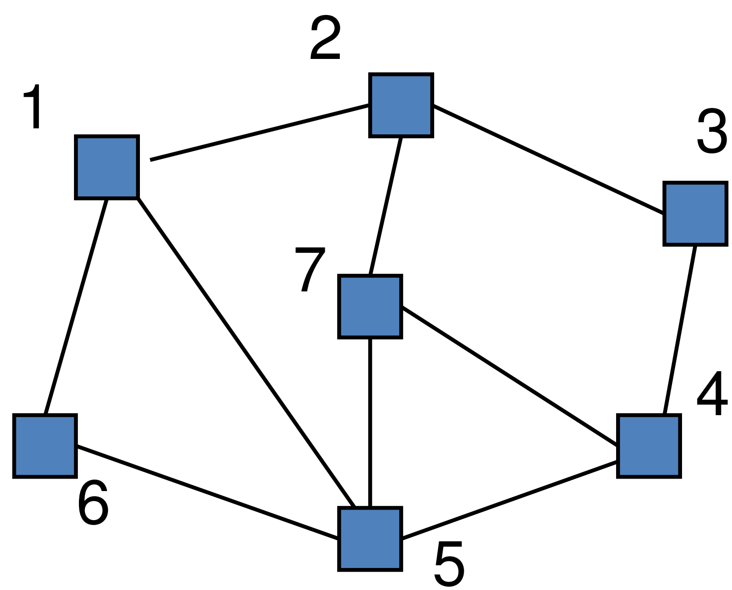
	1	2	3	4	5	6	7
1	0	1	0	0	1	1	0
2	1	0	1	0	0	0	1
3	0	1	0	1	0	0	0
4	0	0	1	0	1	0	1
5	1	0	0	1	0	1	1
6	1	0	0	0	1	0	0
7	0	1	0	1	1	0	0



Implementation: Adjacency Lists

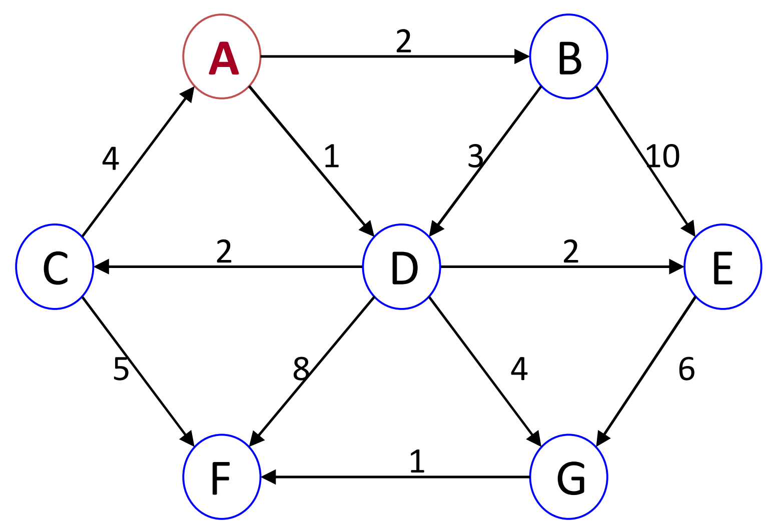
- *n* lists of neighbors;   L[a] = all edges out from v_a
  - Sometimes implemented as a Map<V, List<V>>
  - Good for processing all-neighbors, sparse graphs (few edges)
  - How do we figure out the degree of a vertex?





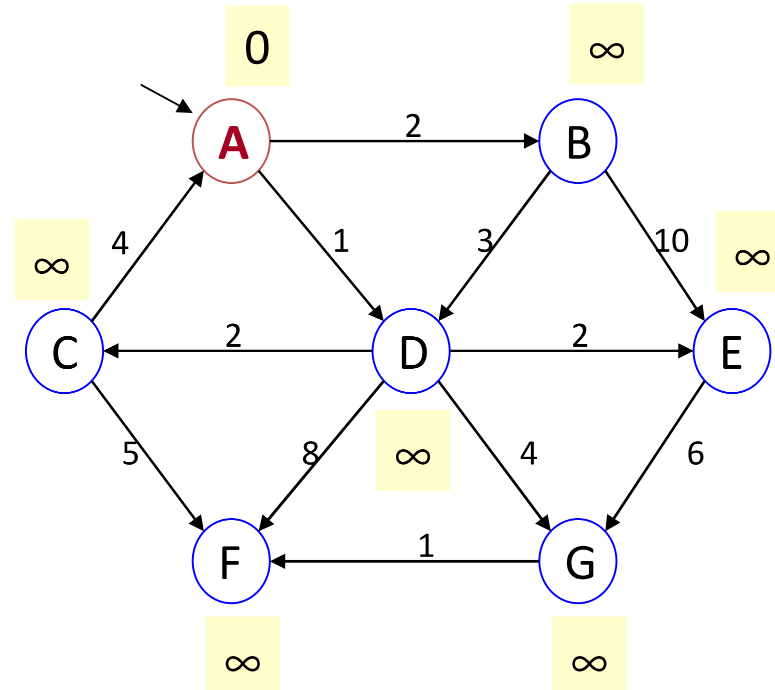
Dijkstra's algorithm

- **Dijkstra's algorithm**: finds shortest (min weight) path between a pair of vertices in a *weighted* directed graph with nonnegative edges
  - solves the "one vertex, shortest path" problem
  - basic algorithm concept: create a table of information about the currently known best way to reach each vertex (distance, previous vertex) and improve it until it reaches the best solution



Dijkstra pseudocode

```
Dijkstra(v1, v2):
   for each vertex v:       // Initialize state
       v's distance := infinity.
       v's previous := none.
   v1's distance := 0.
   Q := {all vertices}.

   while Q is not empty:
       v := remove Q's vertex with min distance.
       mark v as known.
       for each unknown neighbor n of v:
           dist := v's distance + edge (v, n)'s weight.

           if dist is smaller than n's distance:
               n's distance := dist.
               n's previous := v.

   reconstruct path from v2 back to v1,
      following previous pointers.
```

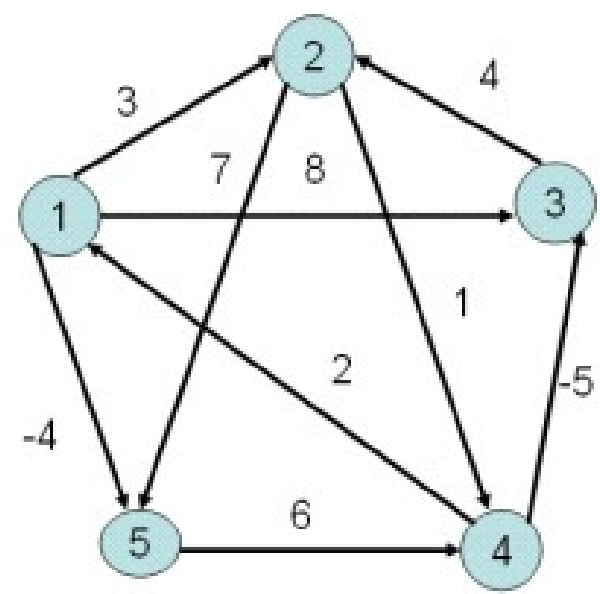


examine A: update B(2),D(1)
examine D: update C(3),E(3),F(9),G(5)
examine B,E: update none
examine C: update F(8)
examine G: update F(6)
examine F: update none

Floyd-Warshall algorithm

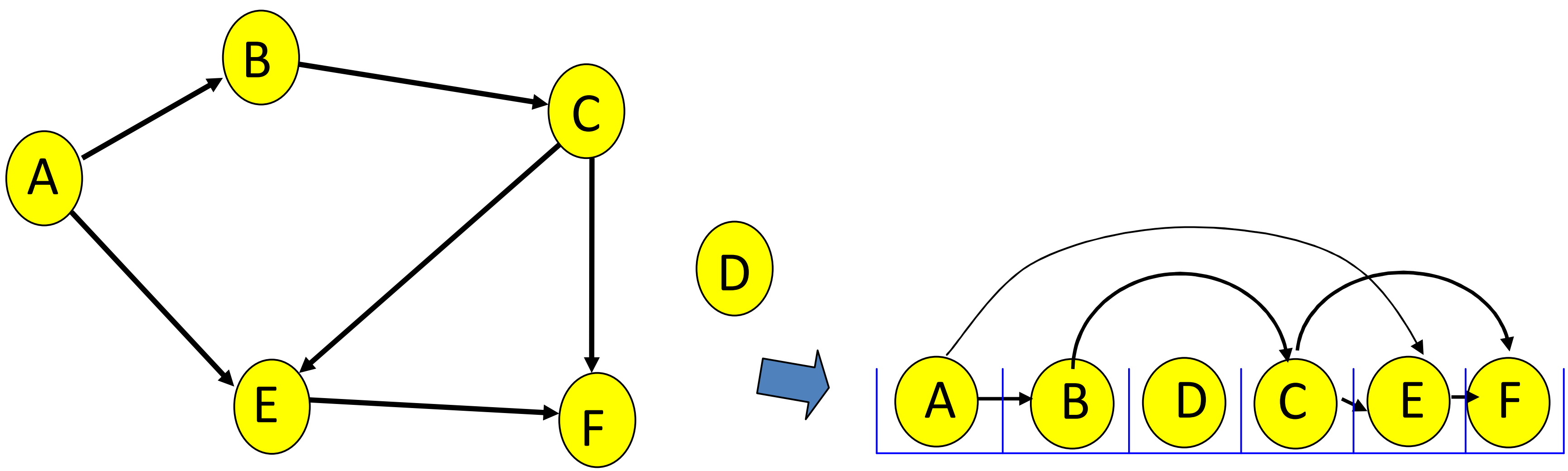
- **Floyd-Warshall algorithm**: finds shortest (min weight) path between *all* pairs of vertices in a weighted directed graph
  - solves the "all pairs, shortest paths" problem (demo)
  - idea: repeatedly find best path using only vertices 1..k inclusive

```
floydWarshall():
   int path[n][n].
   for each (i, j) from (0, 0) to (n, n):
      path[i][j] = edge_weight[i][j].

   for k = 0 to n:
      for i = 0 to n:
         for j = 0 to n:
            path[i][j] = min(path[i][j],
                             path[i][k] + path[k][j]).
```



Topological Sort

- **Topological sort**: finds a total ordering of vertices such that for any edge (v, w) in E, v precedes w in the ordering
  - e.g. find an ordering in which all UW CSE courses can be taken

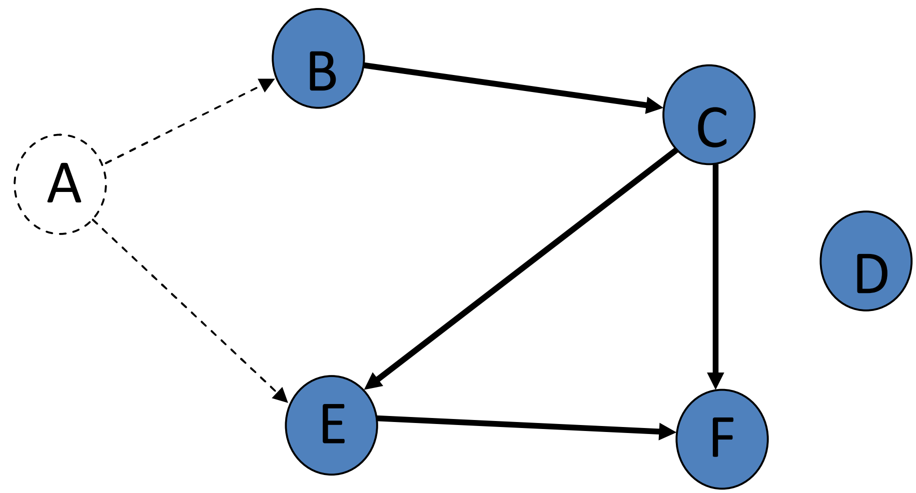


Topological Sort pseudocode

V = {all vertices}.
E = {all edges}.
L = [].

while V is not empty:
    for each vertex v in V:
        if v has **no incoming edges**:
            V.**remove**(v).
            L.**append**(v).
            for each edge e (v → n):
                E.**remove**(e).

return L.



examine A,D
examine B
examine C
examine E
examine F