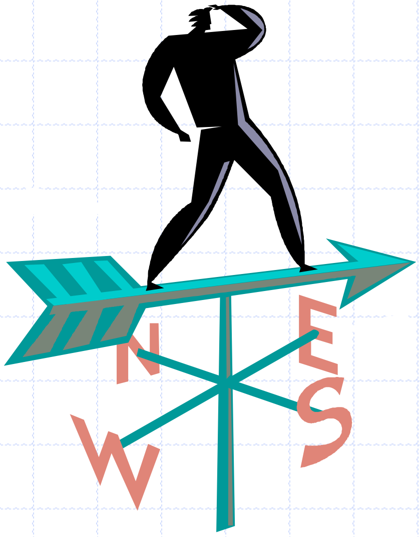
방향그래프

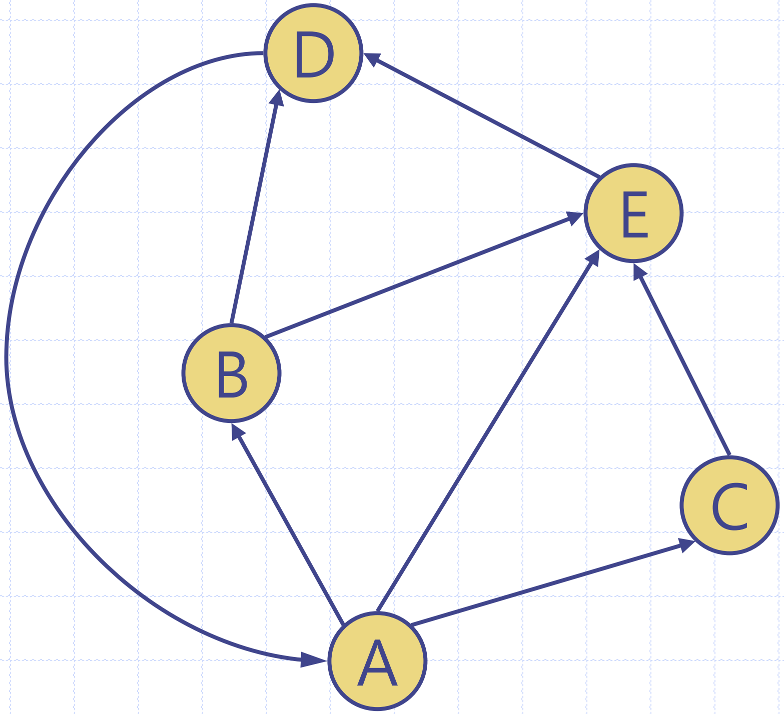


Outline

- 15.1 방향그래프
- 15.2 동적프로그래밍
- 15.3 방향 비싸이클 그래프
- 15.4 응용문제

방향그래프

- **방향그래프**(digraph): 모든 간선이 **방향간선**인 그래프
  - "**directed graph**"의 준말
- **응용**
  - 일방통행 도로
  - 항공노선
  - 작업스케줄링

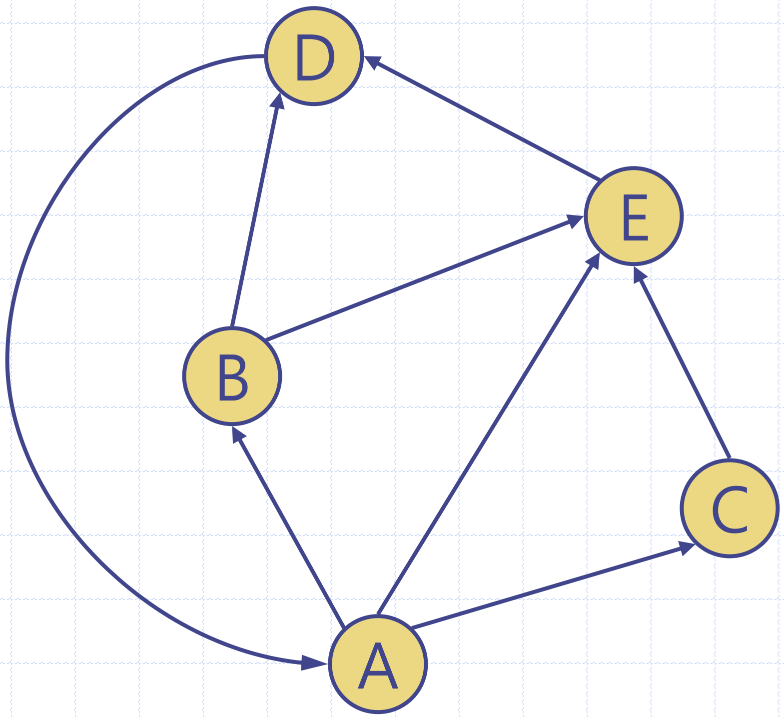


방향그래프 속성

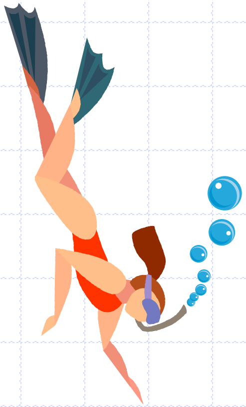
- 모든 간선이 한 방향으로 진행하는 그래프 $G = (V, E)$에서,
  - 간선 (a, b)는 a에서 b로 가지만 b에서 a로 가지는 않는다
- G가 단순하다면,

$$m \leq n(n-1)$$

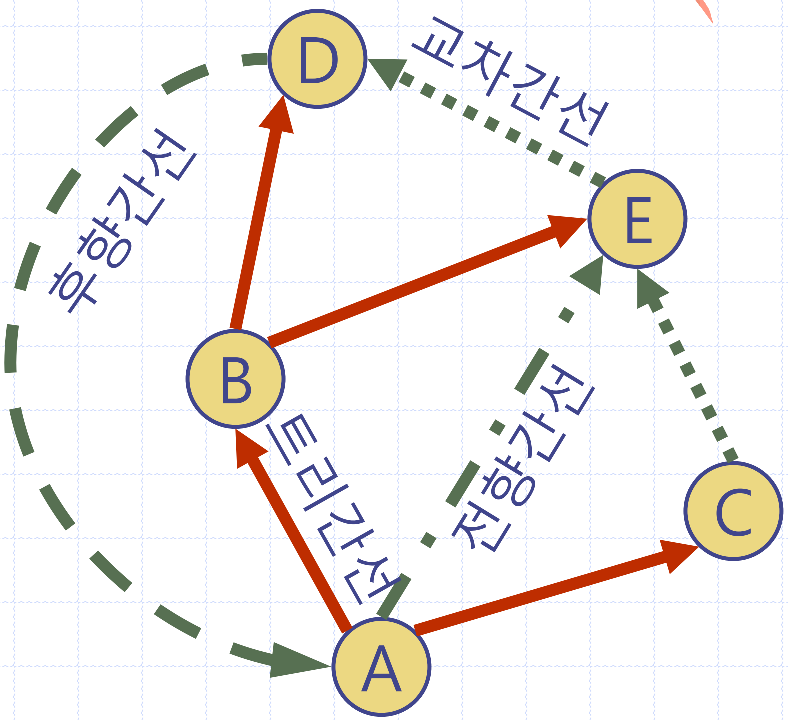
- **진입간선**들(in-edges)과 **진출간선**들(out-edges)을 각각 별도의 인접리스트로 보관한다면, 진입간선들의 집합과 진출간선들의 집합을 각각의 크기에 비례한 시간에 조사 가능

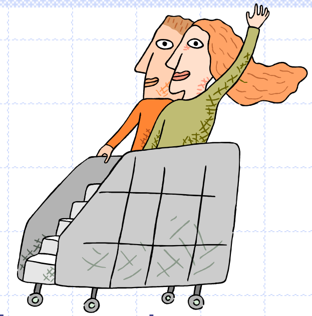


방향 DFS



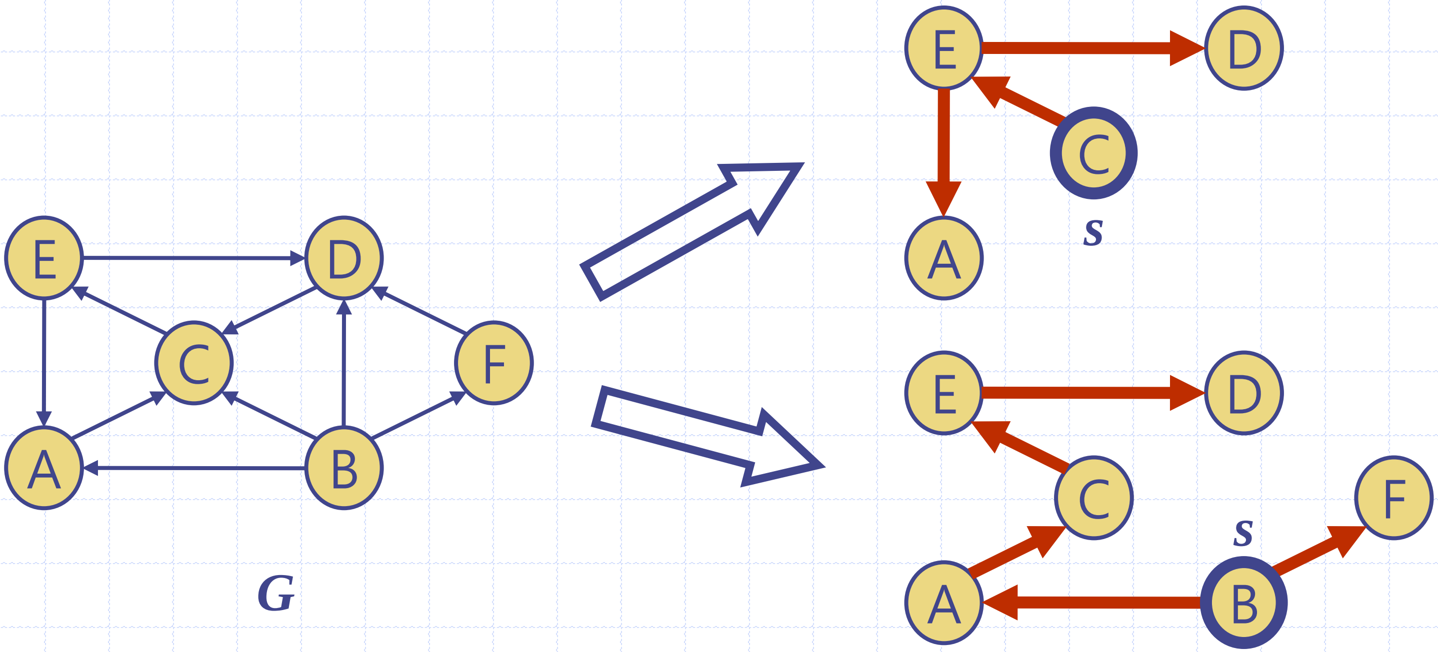
- 간선들을 주어진 방향만을 따라 순회하도록 하면, DFS 및 BFS 순회 알고리즘들을 방향그래프에 특화 가능
- **방향** DFS 알고리즘에서, 네 종류의 간선이 발생
  - **트리간선**(tree edges)
  - **후향간선**(back edges)
  - **전향간선**(forward edges)
  - **교차간선**(cross edges)
- 정점 *s*에서 출발하는 방향 DFS는 *s*로부터 도달 가능한 정점들을 결정





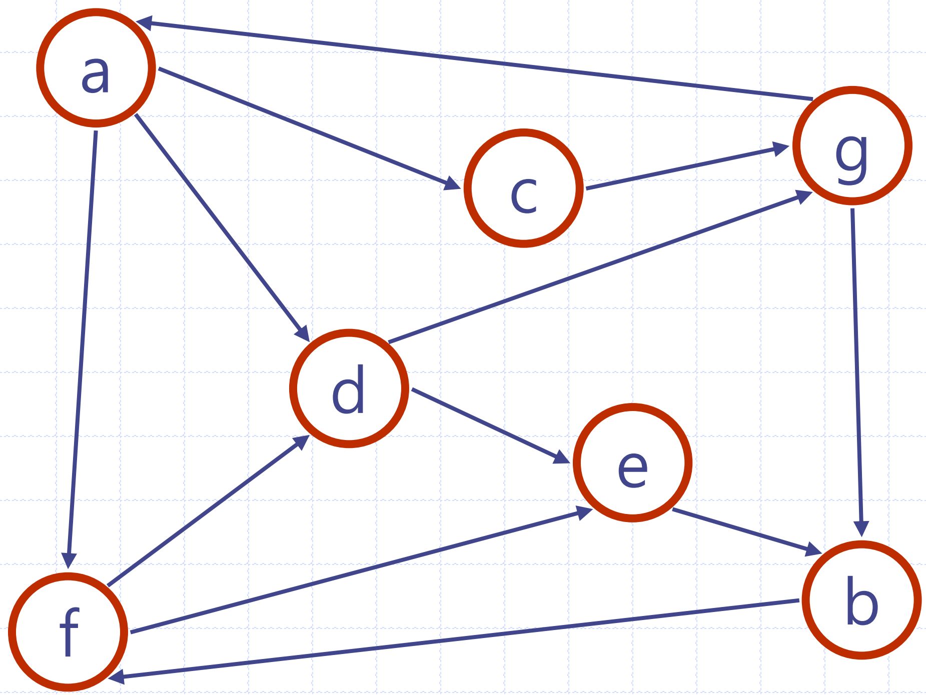
도달 가능성

- 방향그래프 *G*의 두 정점 *u*와 *v*에 대해 만약 *G*에 *u*에서 *v*로의 방향경로가 존재한다면, "*u*는 *v*에 **도달한다**(*u* reaches *v*)", 또는 "*v*는 *u*로부터 **도달 가능**하다(*v* is reachable from *u*)"고 말한다
- *s*를 루트로 하는 **DFS 트리**: *s*로부터 방향경로를 통해 도달 가능한 정점들을 표시



강연결성

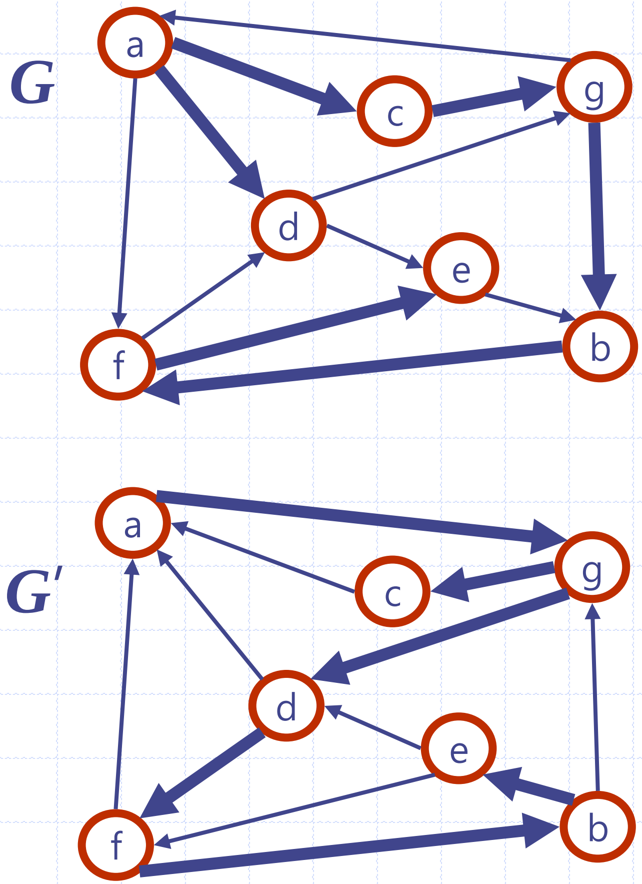
- 방향그래프 G의 어느 두 정점 u와 v에 대해서나 u는 v에 도달하며 v는 u에 도달하면, G를 **강연결**(strongly connected)이라고 말한다 – 즉, 어느 정점에서든지 다른 모든 정점에 도달 가능



강연결 검사 알고리즘

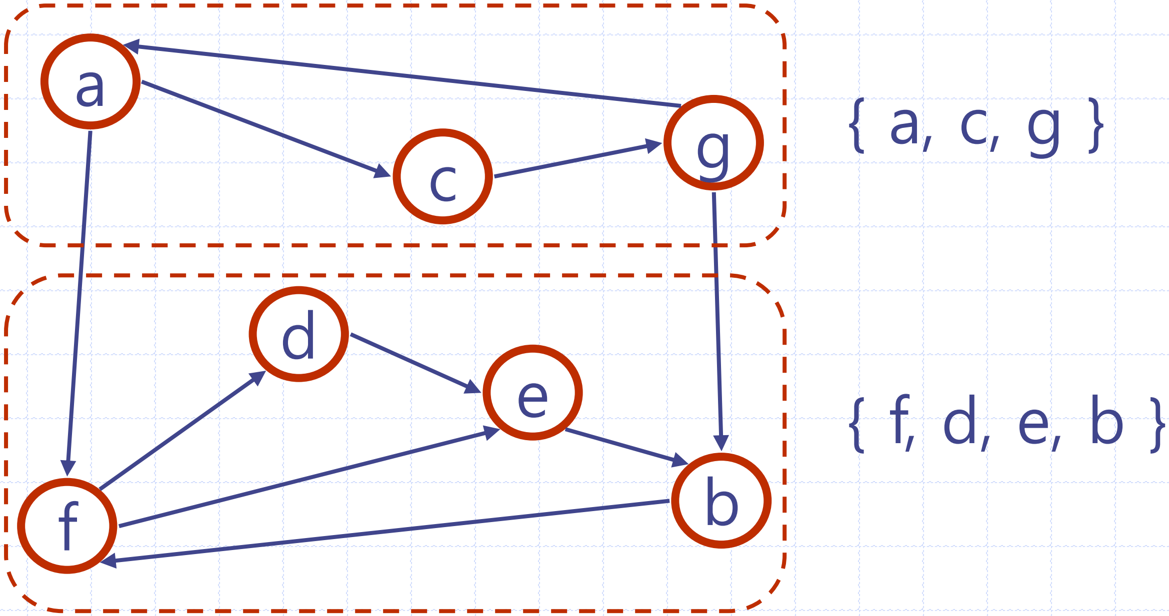
1. *G*의 임의의 정점 *v*를 선택
2. *G*의 *v*로부터 DFS를 수행
    1. 방문되지 않은 정점 *w*가 있다면, ***False***를 반환
3. *G*의 간선들을 모두 역행시킨 그래프 *G′*를 얻음
4. *G′*의 *v*로부터 DFS를 수행
    1. 방문되지 않은 정점 *w*가 있다면, ***False***를 반환
    2. 그렇지 않으면, ***True***를 반환

- **실행시간: O(*n* + *m*)**



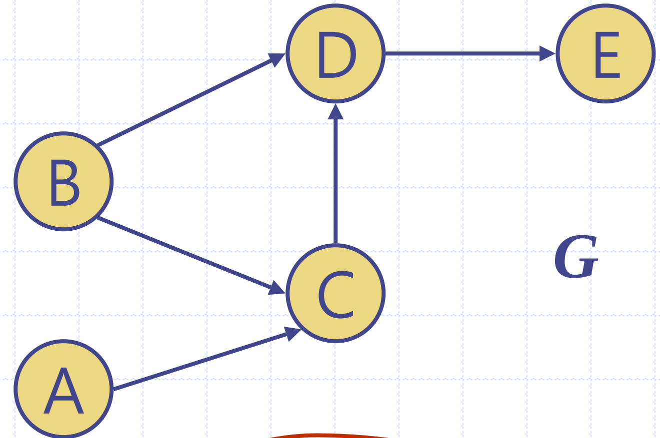
강연결요소

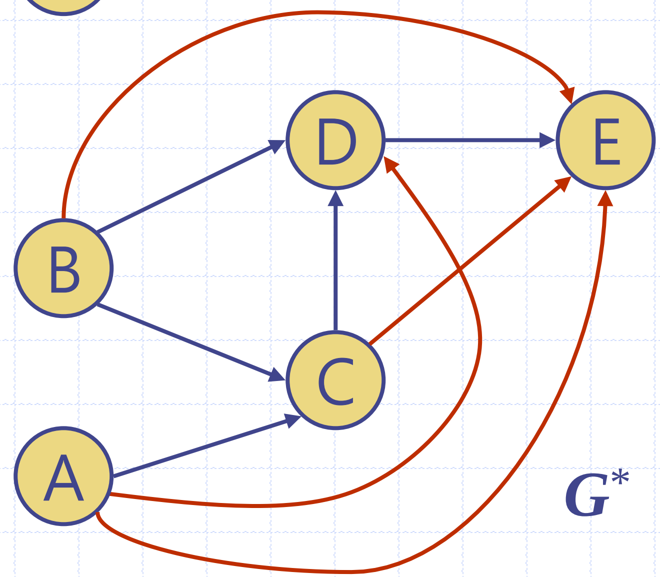
- 방향그래프에서 각 정점으로부터 다른 모든 정점으로 도달할 수 있는 최대의 부그래프
- DFS를 사용하여 $O(n + m)$ 시간에 계산 가능(**biconnectivity**와 유사)



이행적폐쇄

- 주어진 방향그래프 G에 대해, 그래프 G의 **이행적폐쇄**(transitive closure): 다음을 만족하는 방향그래프 G^*
  - G^*는 G와 동일한 정점들로 구성
  - G에 u로부터 $v \neq u$ 로의 방향경로가 존재한다면 G^*에 u로부터 v로의 방향간선이 존재
- **이행적폐쇄**는 방향그래프에 관한 도달 가능성 정보를 제공
- **예:** 컴퓨터 네트워크에서, "노드 a에서 노드 b로 메시지를 보낼 수 있을까?"



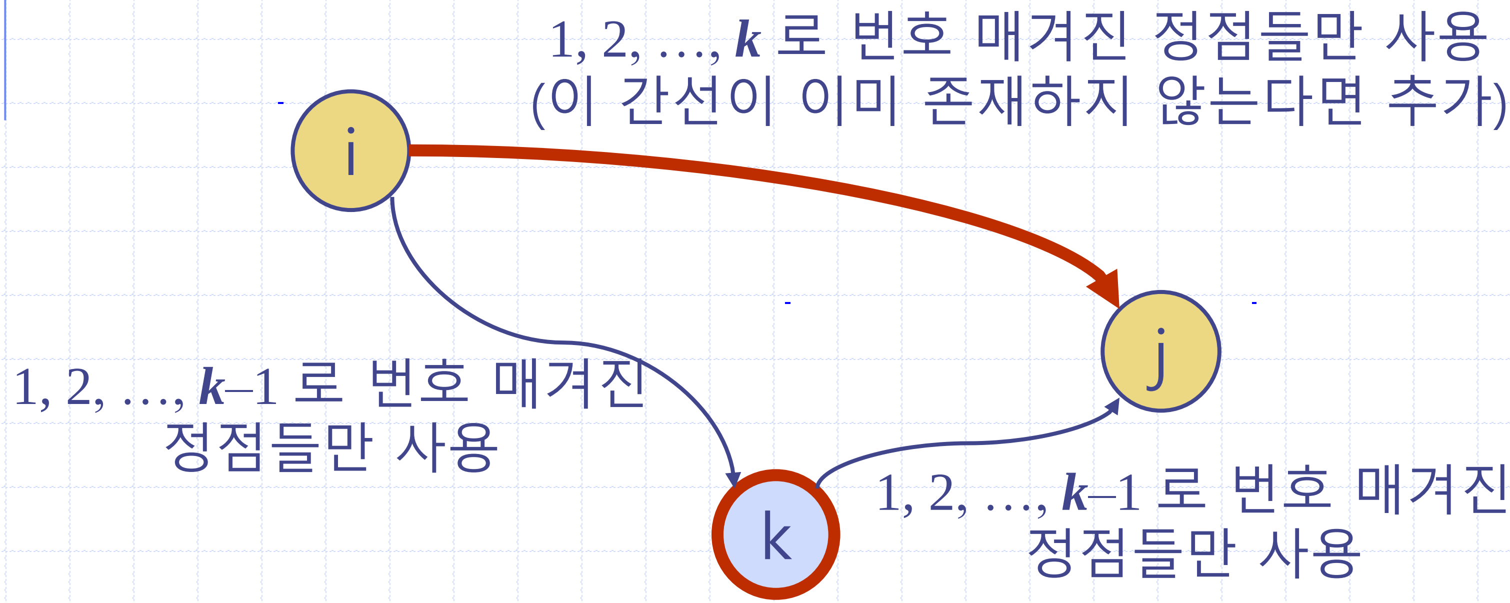


이행적폐쇄 계산

- 각 정점에서 출발하여 DFS를 수행할 수 있다
  - **실행시간:** $O(n(n + m))$
- 대안으로, **동적프로그래밍**(dynamic programming)을 사용할 수 있다: Floyd-Warshall의 알고리즘
  - **원리:** "A에서 B로 가는 길과 B에서 C로 가는 길이 있다면, A에서 C로 가는 길이 있다"

Floyd-Warshall 이행적폐쇄

1. 정점들을 1, 2, …, n 으로 번호를 매긴다
2. 1, 2, …, k 로 번호 매겨진 정점들만 경유 정점으로 사용하는 경로들을 고려



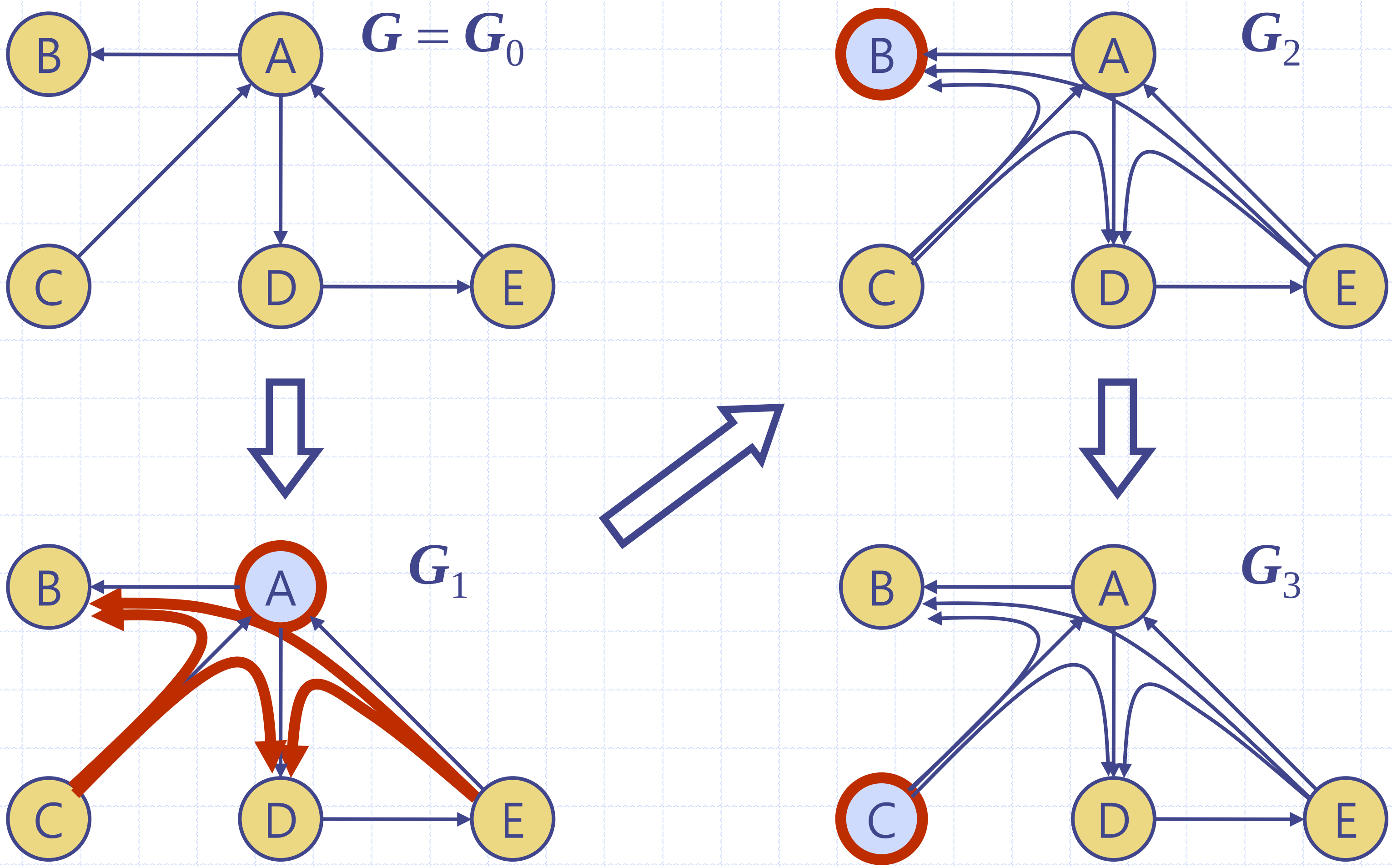
Floyd-Warshall 알고리즘

- Floyd-Warshall 알고리즘은 G의 정점들을 $v_1, \dots, v_n$로 번호 매긴 후, 방향그래프 $G_0, \dots, G_n$을 잇달아 계산
  - $G_0 = G$
  - G에 $\{v_1, \dots, v_k\}$ 집합 내의 경유정점을 사용하는, v_i에서 v_j로의 방향경로가 존재하면 G_k에 방향간선 (v_i, v_j)를 삽입
- k 단계에서, 방향그래프 G_{k-1}로부터 G_k를 계산
- 마지막에 $G_n = G^*$를 얻음
- **실행시간:** $O(n^3)$
  - **전제:** areAdjacent가 $O(1)$ 시간에 수행(즉, 인접행렬)

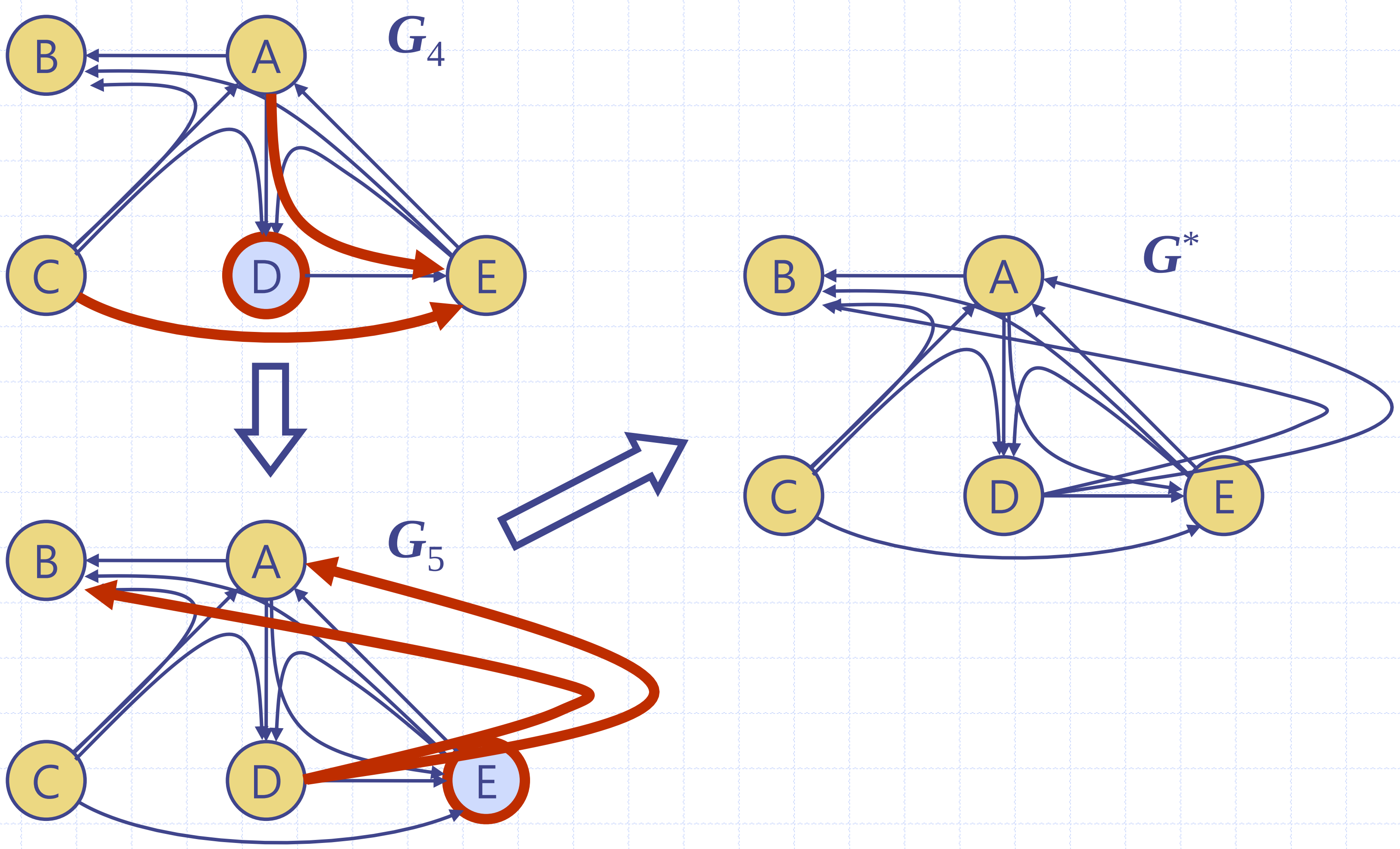
Alg ***Floyd-Warshall*(*G*)**
    input a digraph G with n vertices
    output transitive closure G^* of G

1. *Let* $v_1, v_2, \dots, v_n$ *be an arbitrary numbering of the vertices of* G
2. $G_0 \leftarrow G$
3. **for** $k \leftarrow 1$ **to** n {stopover vertex}
   - $G_k \leftarrow G_{k-1}$
   - **for** $i \leftarrow 1$ **to** n, $i \neq k$ {start vertex}
     - **for** $j \leftarrow 1$ **to** n, $j \neq i, k$ {end vertex}
       - **if** (G_{k-1}.*areAdjacent*(v_i, v_k) & G_{k-1}.*areAdjacent*(v_k, v_j))
         - **if** (!G_k.*areAdjacent*(v_i, v_j))
           - G_k.*insertDirectedEdge*(v_i, v_j, k)
4. **return** G_n

Floyd-Warshall 알고리즘 수행 예



Floyd-Warshall 알고리즘 수행 예 (conti.)



동적프로그래밍

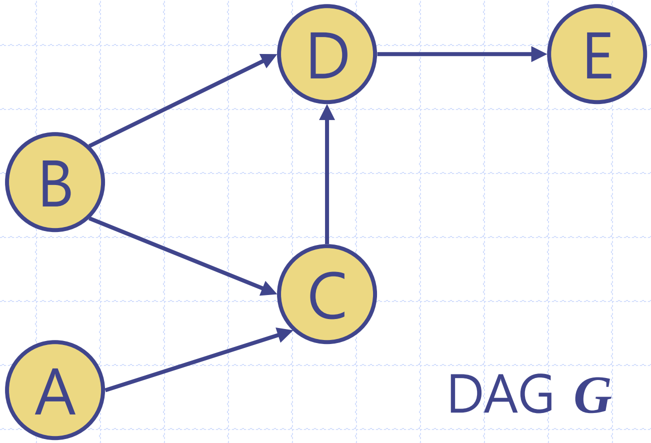
- **동적프로그래밍**(dynamic programming): 알고리즘 설계의 일반적 기법 가운데 하나
- 언뜻 보기에 많은 시간(심지어 지수 시간)이 소요될 것 같은 문제에 주로 적용 – 적용의 조건은:
  - **부문제 단순성**(simple subproblems): 부문제들이 j, k, l, m, 등과 같은 몇 개의 변수로 정의될 수 있는 경우
  - **부문제 최적성**(subproblem optimality): 전체 최적치가 최적의 부문제들에 의해 정의될 수 있는 경우
  - **부문제 중복성**(subproblem overlap): 부문제들이 독립적이 아니라 상호 겹쳐질 경우 – 따라서, 해가 "상향식"으로 구축되어야 함
- **예**
  - 피보나치 수열(Fibonacci progression)에서 n-번째 수 찾기
  - 그래프의 이행적폐쇄 계산하기

동적프로그래밍 vs. 분할통치법

- 공통점
  - 알고리즘 설계기법의 일종
  - 문제공간: **원점-목표점** 구조
    - **원점:** 문제의 초기 또는 기초 지점(복수 개수 가능)
    - **목표점:** 최종해가 요구되는 지점 (보통 1개)
    - 추상적 개념 상의 두 지점
- 차이점
  - 문제해결 진행 방향
    - **동적프로그래밍**(단방향): 원점 ⇒ 목표점
    - **분할통치**(양방향): 목표점 ⇒ 원점 ⇒ 목표점 (단, **해**를 구하기 위한 연산 진행 방향은 원점 ⇒ 목표점)
- 성능
  - **동적프로그래밍**
    - 단방향 특성때문에 종종 효율적
  - **분할통치**
    - 분할 회수
    - 중복연산 수행 회수

방향 비싸이클 그래프

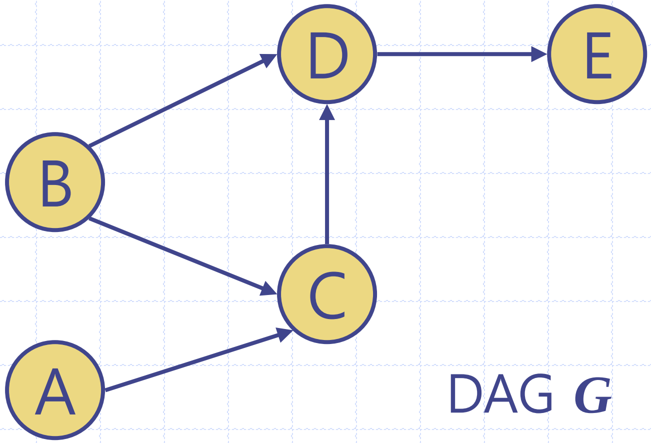
- **방향 비싸이클 그래프**(directed acyclic graph, **DAG**): 방향싸이클이 존재하지 않는 방향그래프
- **예**
  - C++ 클래스 간의 상속 또는 Java 인터페이스
  - 교과목 간의 선수 관계
  - 프로젝트의 부분 작업들 간의 스케줄링 제약
  - 사전의 용어 간의 상호의존성
  - 엑셀과 같은 스프레드시트에서 수식 간의 상호의존성



DAG G

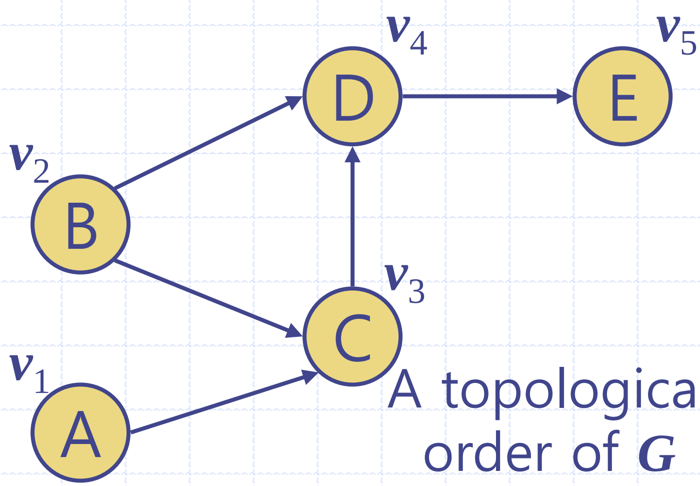
DAG와 위상 정렬

- 방향그래프의 **위상순서**(topological order)
  - 모든 $i < j$인 간선 (v_i, v_j)에 대해 정점들을 번호로 나열한 것
  - **예:** 작업스케줄링 방향그래프에서 위상순서는 작업들의 우선 순서 제약을 만족하는 작업 순서



DAG G

정리: 방향그래프가 DAG면, 위상순서를 가지며, 그 역도 참이다



A topological order of G

위상 정렬

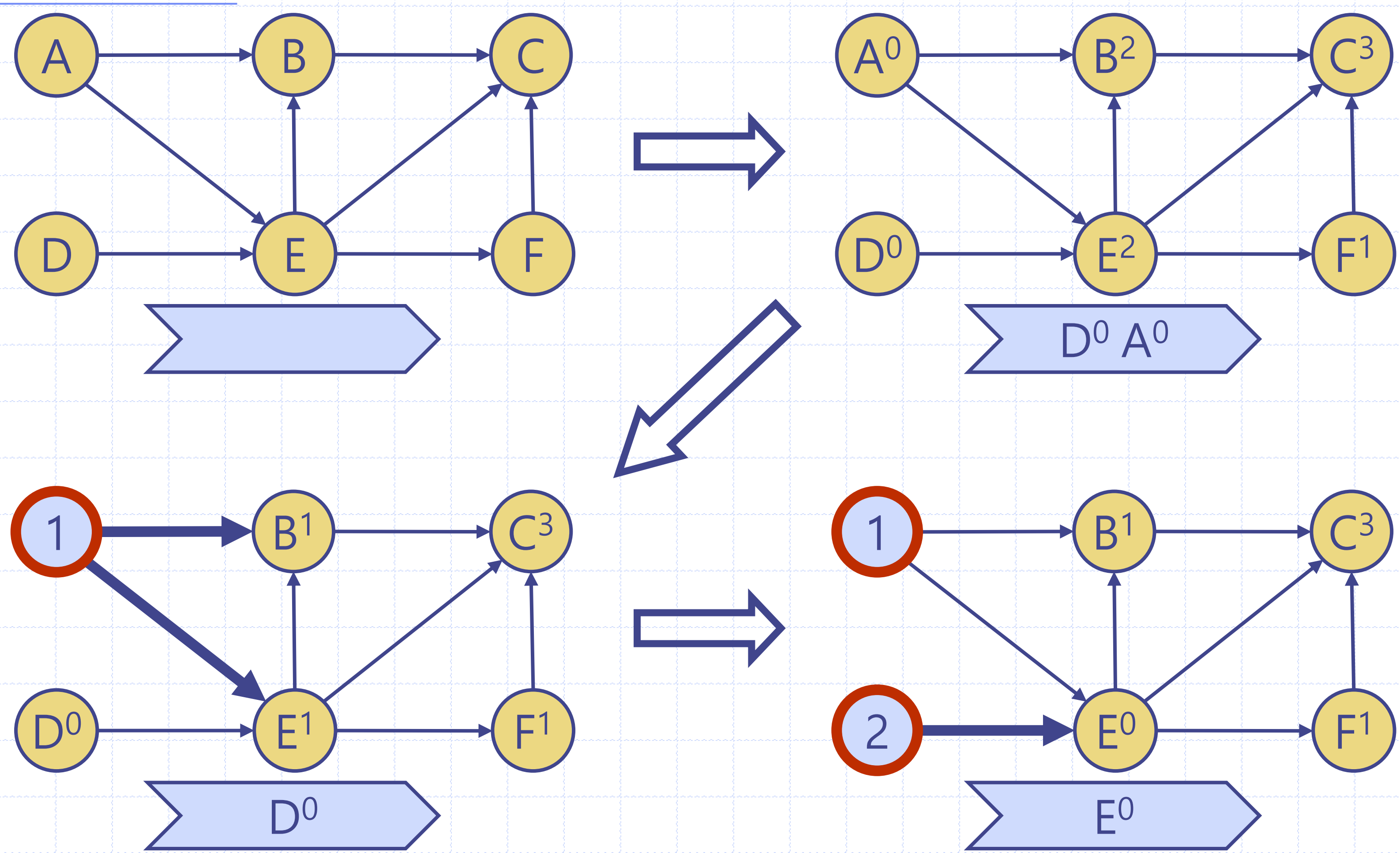
- **위상 정렬**(topological sort): DAG로부터 위상순서를 얻는 절차
- 알고리즘
  - topologicalSort: 정점의 진입차수(in-degree)를 이용
  - topologicalSortDFS: DFS의 특화

정점의 진입차수를 이용하는 위상 정렬

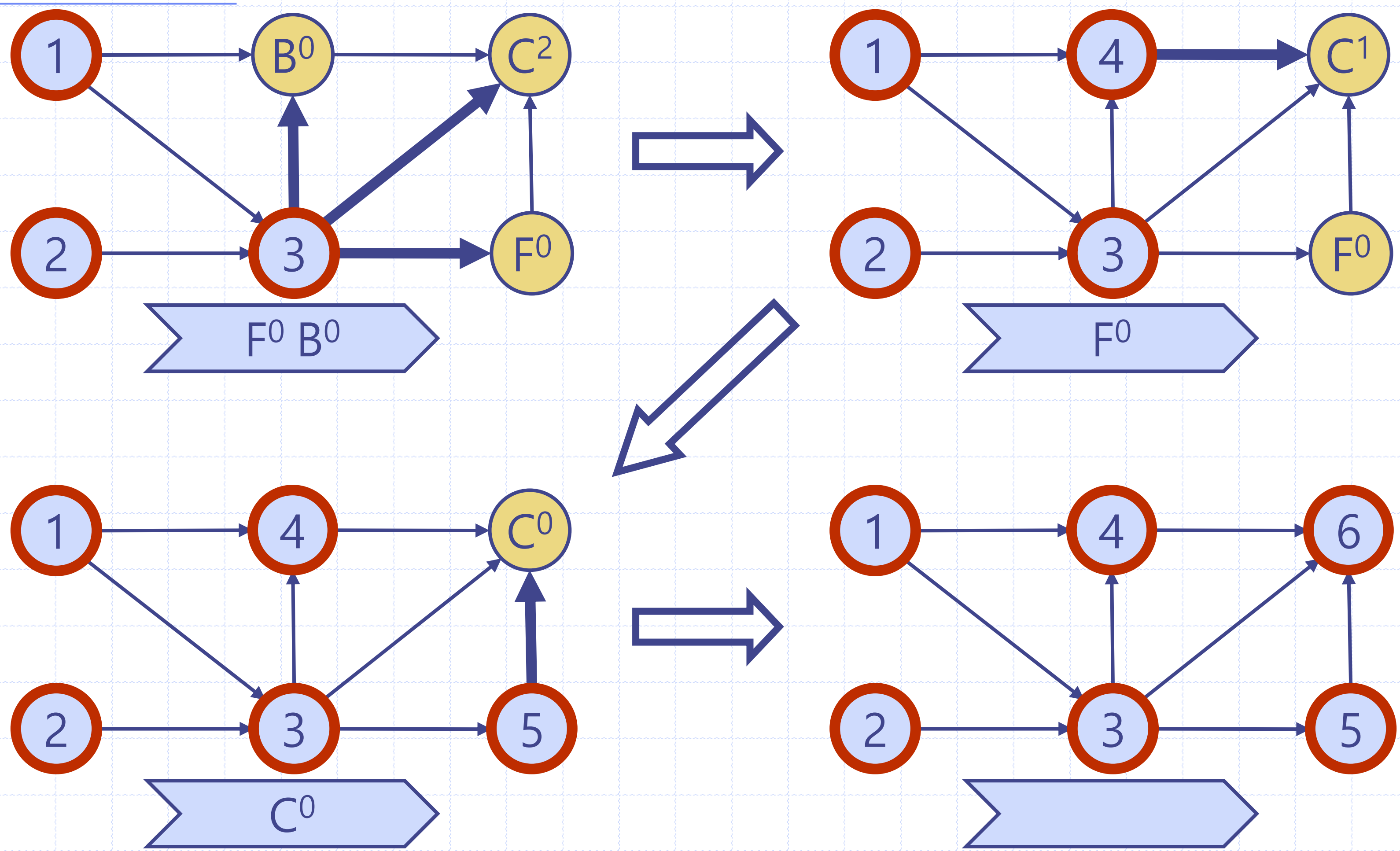
```
Alg topologicalSort(G)
    input a digraph G with n
        vertices
    output a topological ordering
        v1 , …, vn of G, or an
        indication that G has a
        directed cycle

1. Q ← empty queue
2. for each u ∈ G.vertices()
        in(u) ← inDegree(u)
        if (in(u) = 0)
            Q.enqueue(u)
3. i ← 1          {topological number}
4. while (!Q.isEmpty())
        u ← Q.dequeue()
        Label u with topological number i
        i ← i + 1
        for each e ∈ G.outIncidentEdges(u)
            w ← G.opposite(u, e)
            in(w) ← in(w) – 1
            if (in(w) = 0)
                Q.enqueue(w)
5. if (i ≤ n)     {i = n + 1, for DAG}
        write("G has a directed cycle")
6. return
```

- 각 정점에 새로운 라벨을 정의
  - **현재 진입차수**(inCount): $in(v)$, 정점 v의 현재의 진입차수

위상 정렬 수행 예



위상 정렬 수행 예 (conti.)



DFS를 특화한 위상 정렬

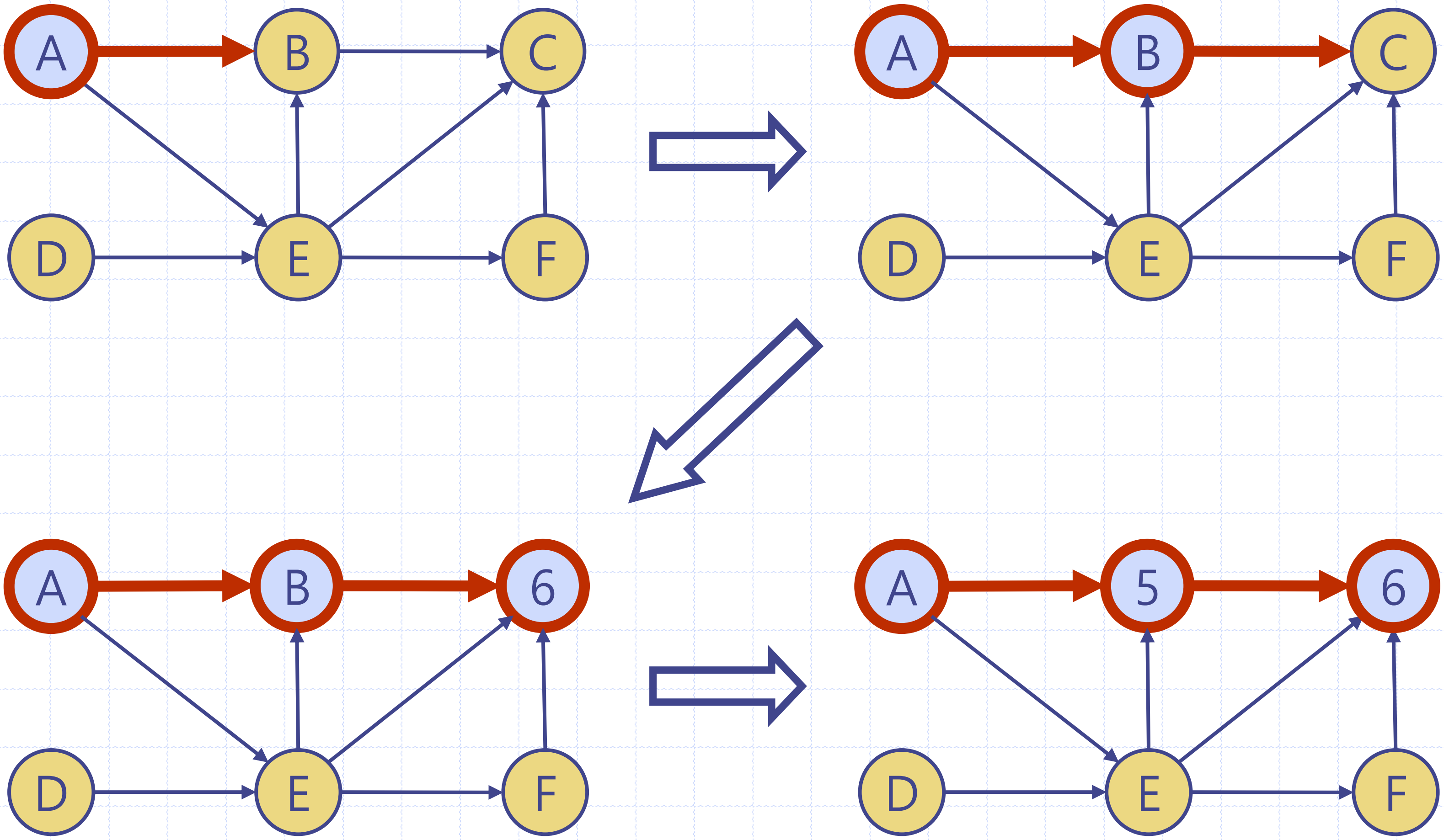
Alg *topologicalSortDFS(G)*
input dag ***G***
output topological ordering of ***G***

1. ***n*** ← ***G.numVertices***()
2. **for each *u*** ∈ ***G.vertices***()
   l(***u***) ← ***Fresh***
3. **for each *v*** ∈ ***G.vertices***()
   if (***l***(***v***) = ***Fresh***)
     rTopologicalSortDFS(***G***, ***v***)

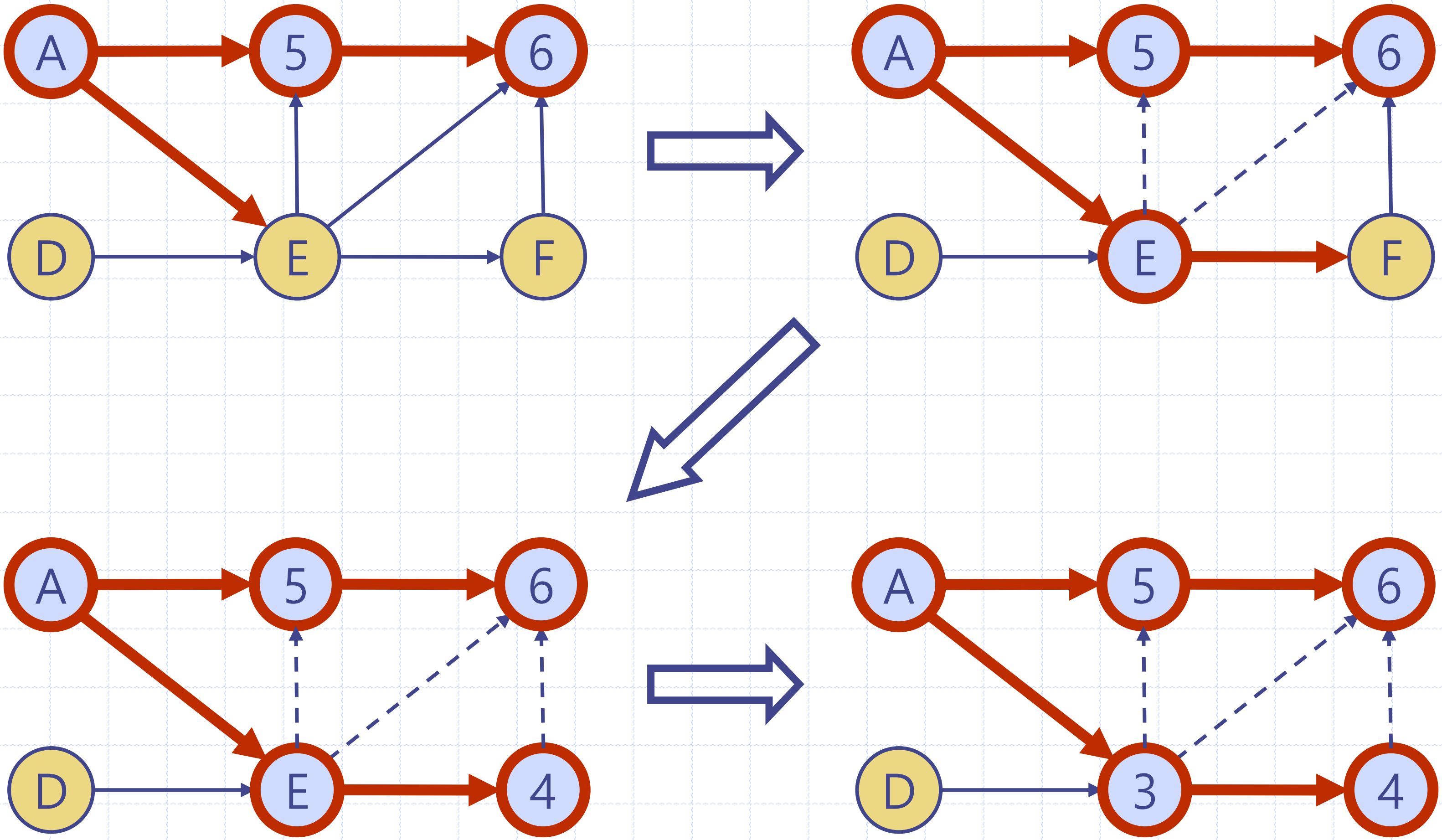
Alg *rTopologicalSortDFS(G, v)*
input graph ***G***, and a start vertex ***v*** of ***G***
output labeling of the vertices of ***G*** in the connected component of ***v***

1. ***l***(***v***) ← ***Visited***
2. **for each *e*** ∈ ***G.outIncidentEdges***(***v***)
   w ← ***opposite***(***v***, ***e***)
   if (***l***(***w***) = ***Fresh***) {***e*** is a tree edge}
     rTopologicalSortDFS(***G***, ***w***)
   elseif *w* *is not labelled with a topological number*
     write(*"G has a directed cycle"*)
   {**else**
     e is a nontree edge}
3. *Label* ***v*** *with topological number* ***n***
4. ***n*** ← ***n*** – 1

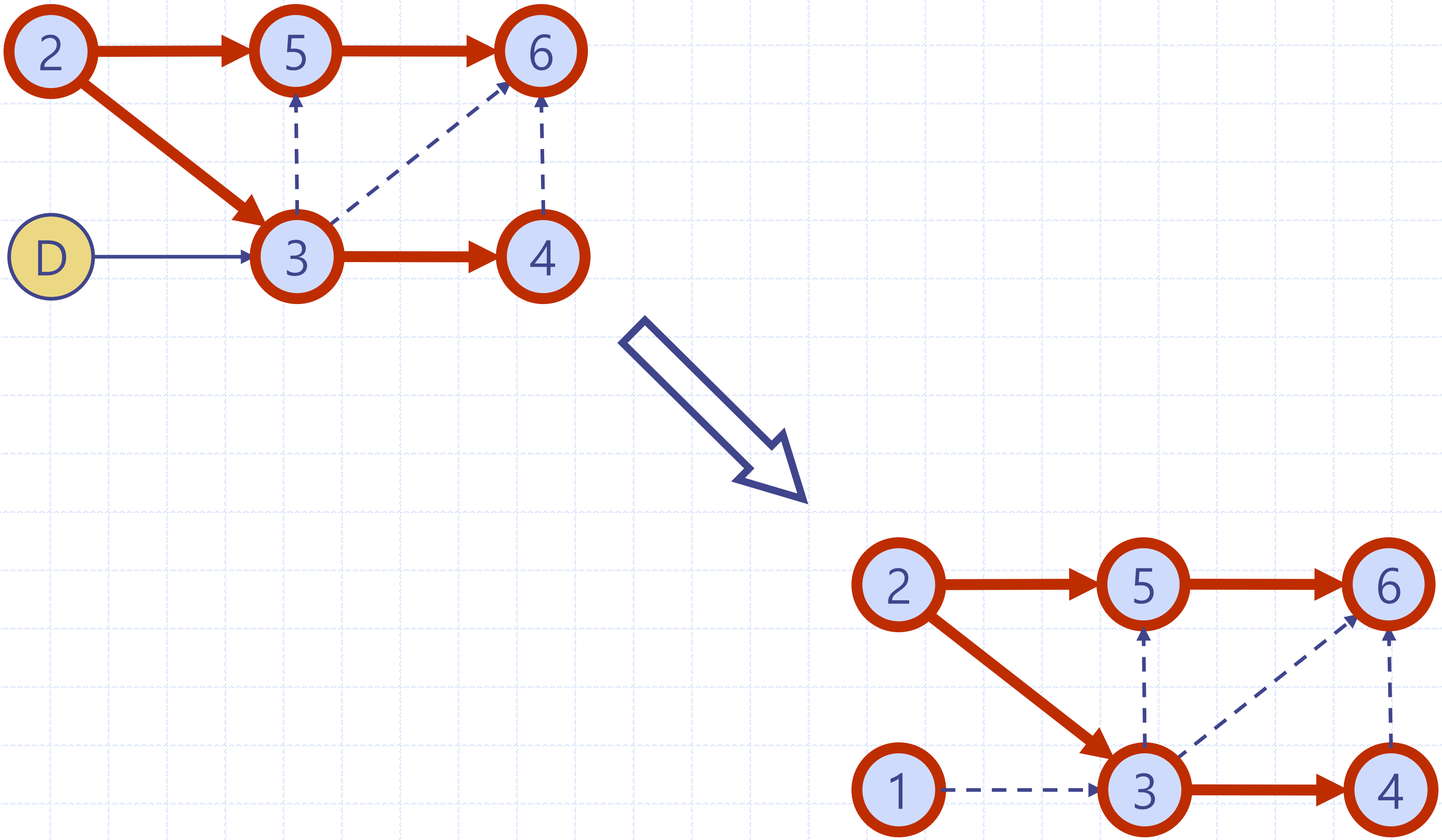
위상 정렬 수행 예



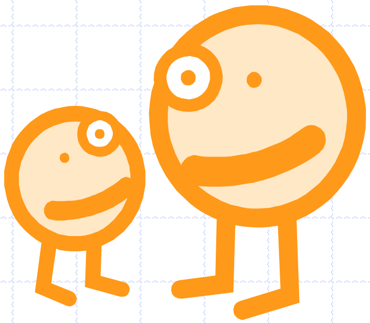
위상 정렬 수행 예 (conti.)



위상 정렬 수행 예 (conti.)



위상 정렬 알고리즘 분석



- 두 개의 버전 모두
  - **O**(n + m) 시간과 **O**(n) 공간 소요
  - G가 DAG인 경우, G의 위상순서를 계산
  - G에 방향싸이클이 존재할 경우, 일부 정점의 순위를 매기지 않은 채로 정지