파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

비순환 방향 그래프(Directed Acyclic Graph)를 이해하고 위상(Topological) 정렬을 학습한다

Data Structures in Python
Chapter 9

- Graph Introduction
- Graph Traversal – BFS
- Graph Traversal – DFS
- **Topological Sort of DAG**

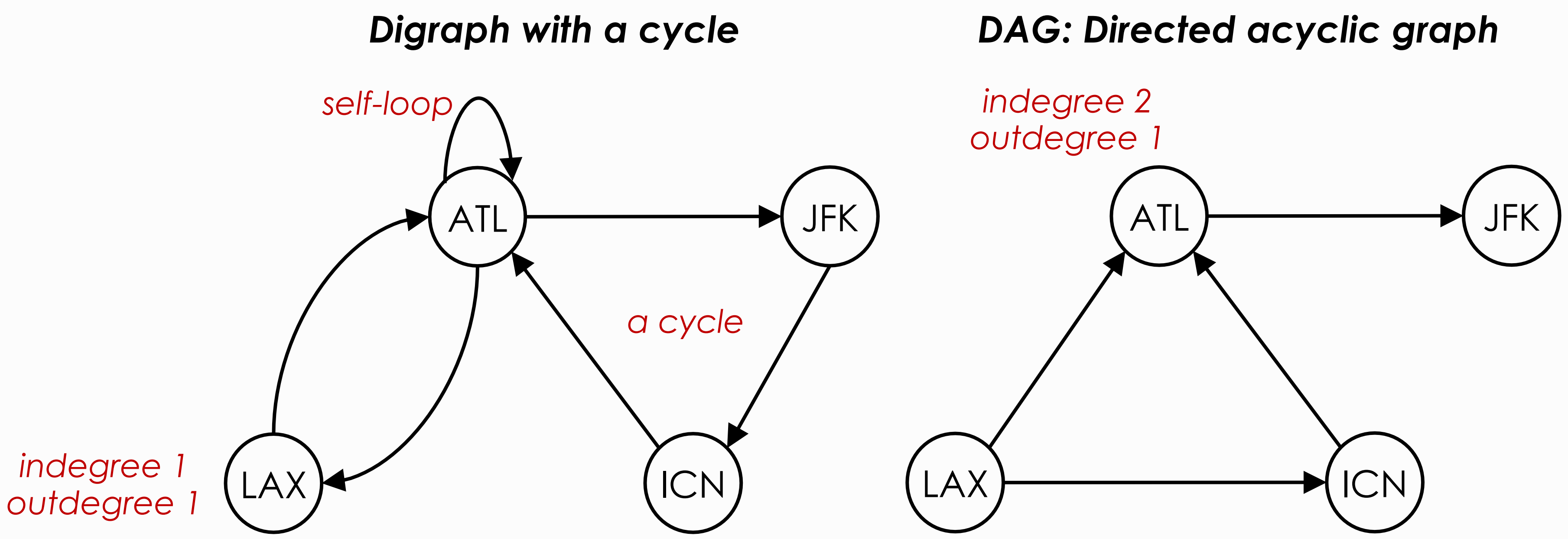
Agenda

- Topological Sort
  - Digraph ADT and Class
  - Digraph DFS and BFS
  - Topological Sort of DAG

- References:
  - Problem Solving with Algorithms and Data Structures
  - Wikipedia: Directed graph
  - Wikipedia: Directed acyclic graph

Digraph Introduction

- When a graph has an **ordered pair** of vertices, it is called a directed graph or **digraph**.
- A special case of **digraph** that contains no cycles is called a **directed acyclic graph**, or **DAG**.
- The **outdegree** of a vertex is the number of edges pointing from it.
- The **indegree** of a vertex is the number of edges pointing to it.



Graph ADT - Review

- Graph processing algorithms first build an internal representation of a graph by adding edges, then process it by iterating through the vertices and through edges that are adjacent to a vertex.

Operations	Description
`g = Graph()`	construct a new Graph object g
`g.addEdge(v, w)`	add two edges v-w and w-v to g for undirected
`g.countV()`	the number of vertices in g
`g.countE()`	the number of edges in g
`g.degree(v)`	the number of neighbors of v in g
`g.hasVertex(v)`	is v a vertex in g?
`g.hasEdge(v, w)`	is v-w an edge in g?
`g.vertices()`	an iterable for the vertices of g
`g.neighbors(v)`	an iterable for the neighbors of vertex v in g
`str(g)`	string representation of g

Graph ADT - Review

- Graph processing algorithms first build an internal representation of a graph by adding edges, then process it by iterating through the vertices and through edges that are adjacent to a vertex.

Operations	Description	
`g = Graph()`	construct a new Graph object g	*Digraph()*
`g.addEdge(v, w)`	add two edges v-w and w-v to g for undirected	*override to add one edge v-w to g*
`g.countV()`	the number of vertices in g	
`g.countE()`	the number of edges in g	
`g.degree(v)`	the number of neighbors of v in g	*inDegree(v), outDegree(v)*
`g.hasVertex(v)`	is v a vertex in g?	
`g.hasEdge(v, w)`	is v-w an edge in g?	
`g.vertices()`	an iterable for the vertices of g	
`g.neighbors(v)`	an iterable for the neighbors of vertex v in g	*inNeighbors(v), outNeighbors(v)*
`str(g)`	string representation of g	

Graph ADT for Digraph

- Graph processing algorithms first build an internal representation of a graph by adding edges, then process it by iterating through the vertices and through edges that are adjacent to a vertex.

Operations	Description	
`g = Digraph()`	construct a new **Digraph** object g	*Digraph()*
`g.addEdge(v, w)`	add one edge **v-w** to g	*override to add one edge v-w to g*
`g.countV()`	the number of vertices in g	
`g.countE()`	the number of edges in g	
`g.inDegree(v)` **`g.degree(v)`**	the number of **incoming** neighbors of v in g the number of **outgoing** neighbors of v in g	*add inDegree(v)*
`g.hasVertex(v)`	is v a vertex in g?	
`g.hasEdge(v, w)`	is v-w an edge in g?	
`g.vertices()`	an iterable for the vertices of g	
`g.inNeighbors(v)` **`g.neighbors(v)`**	an iterable for **incoming** neighbors of vertex v in g an iterable for **outgoing** neighbors of vertex v in g	*add inNeighbors(v)*
`str(g)`	string representation of g	

Digraph Class

- Define a **`Digraph`** class derived from **`Graph`** class.
- Override **`addEdge(v, w)`** method.
- Define **`inDegree(v)`** and **`inNeighbors(v)`** for incoming vertices **to v.**
- To use **`DFS()`, `IDFS()`**, and **`BFS()`** as they are,
  do not define **`outDegree(v)`** and **`outNeighbors(v)`** for outgoing vertices **from v.**

```
class Digraph(Graph):
    def addEdge(self, v, w):
        pass

    def inDegree(self, v):
        pass

    def inNeighbors(self, v):
        pass

    def outDegree(self, v):
        return super().degree(v)

    def outNeighbors(self, v):
        return super().neighbors(v)
```

override to add one edge v-w to g

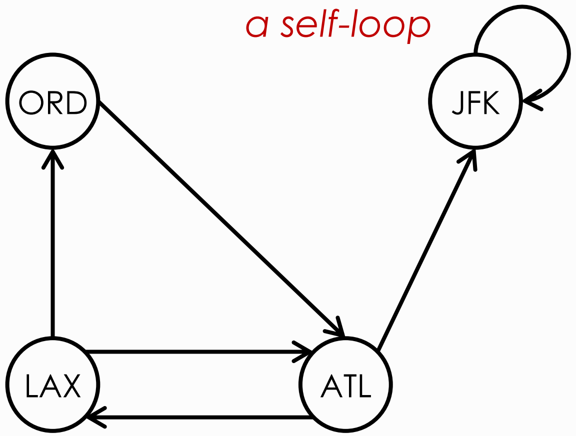
*return **len**([list of incoming vertices])*

*return **iter**([list of incoming vertices])*

Leave out these two definitions, then
*existing **degree(v)** and **neighbors(v)***
will work for outgoing vertices from v.

Digraph Class DFS

- To compute the DFS for a digraph, we may use the same code used for undirected graphs.



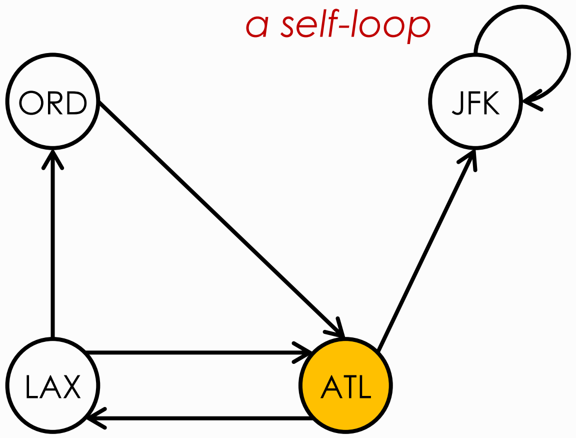
Digraph with a self-loop

Adjacency list

```
LAX: ORD ATL
ORD: ATL
ATL: LAX JFK
JFK: JFK
```

Digraph Class DFS

- To compute the DFS for a digraph, we may use the same code used for undirected graphs.



Digraph with a self-loop

Adjacency list

```
LAX: ORD ATL
ORD: ATL
ATL: LAX JFK
JFK: JFK
```

Pseudocode

```
def DFS(g, v):
    path.append(v)
    for w in g.neighbors(v):
        if w not in path:
            DFS(g, w)
```

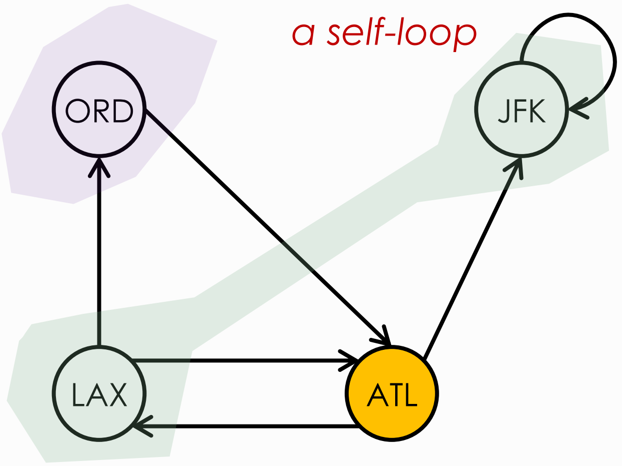
Sample Run: DFS at ATL

```
['ATL', 'LAX', 'ORD', 'JFK']
```

We start traversal from vertex ATL. Look for neighbors, **LAX** and JFK. Then go for LAX first (why?). Look for all neighbors, ATL and ORD. Since ATL is a visited vertex, now go for ORD. Now it reached a dead-end since it started ATL→LAX. Now back-track and found ATL → JFK not visited. Now go for JFK.
A Depth-first traversal of the graph above is ATL, LAX, ORD, JFK.

Digraph Class BFS - Review

- Just like DFS, the same BFS algorithm for the undirected graph works for digraphs.

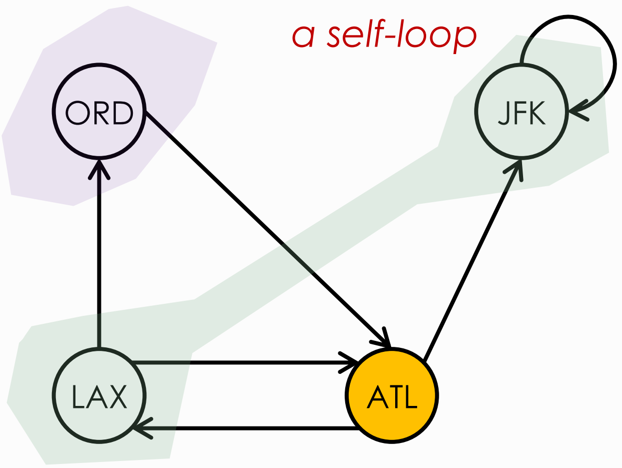


Digraph with a self-loop

```
class BFS:
    def __init__(self, graph, s):
        self._distTo = dict()
        self._prevTo = dict()

        self._distTo[s] = 0
        self._prevTo[s] = None
        self._path = []

        queue = deque()
        queue.append(s)

        while queue:
            v = queue.popleft()
            for w in g.neighbors(v):
                if w not in self._distTo:
                    queue.append(w)
                    self._distTo[w] = 1 + self._distTo[v]
                    self._prevTo[w] = v
```

Digraph Class BFS

- Just like DFS, the same BFS algorithm for the undirected graph works for digraphs.



Digraph with a self-loop

Adjacency list

```
LAX: ORD ATL
ORD: ATL
ATL: LAX JFK
JFK: JFK
```

```
class BFS:
    def __init__(self, graph, s):
        self._distTo = dict()
        self._prevTo = dict()

        self._distTo[s] = 0
        self._prevTo[s] = None
        self._path = []

        queue = deque()
        queue.append(s)

        while queue:
            v = queue.popleft()
            for w in g.neighbors(v):
                if w not in self._distTo:
                    queue.append(w)
                    self._distTo[w] = 1 + self._distTo[v]
                    self._prevTo[w] = v
```

Sample Run: BFS at ATL

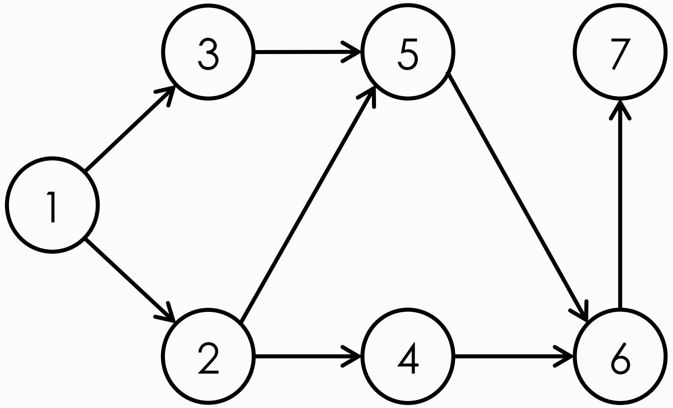
```
[ATL, LAX, JFK, ORD]
```

Digraph DFS/BFS Exercise

- Perform DFS and BFS for all vertices in the following digraph.

1234567.txt
edge list

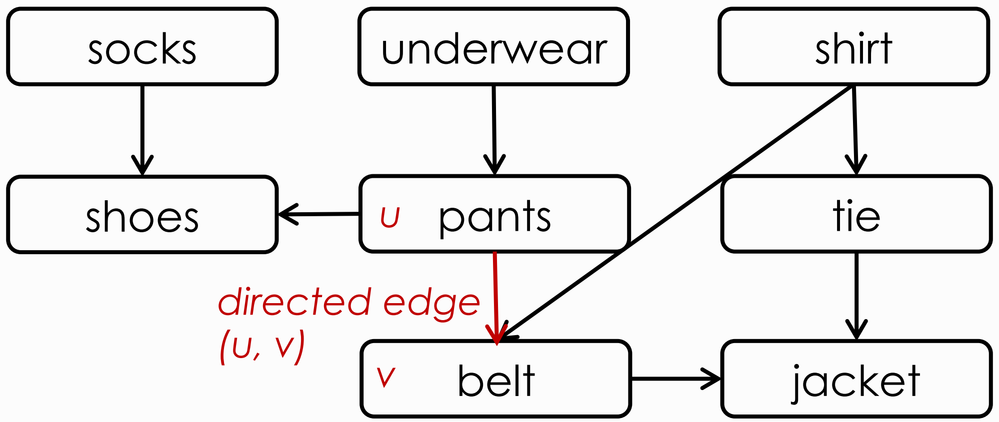
```
1 2
1 3
2 4
2 5
3 5
4 6
5 6
6 7
```




```
recursive DFS: 1,                3
iterative DFS: 1,                4
          BFD: 1,                7
```

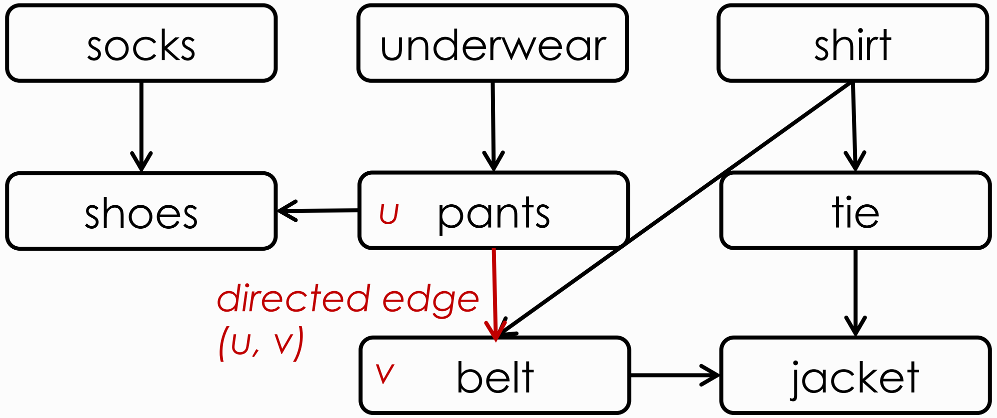
Topological Sort of DAG(위상 정렬)

- A directed edge in DAG is often used to indicate precedence among events.
  - For example, consider how a person might wear several garments, such as socks, pants, shoes, etc. Some must be put on before the others. We need to **wear pants u before belt v**.



Topological Sort of DAG(위상 정렬)

- A directed edge in DAG is often used to indicate precedence among events.
  - For example, consider how a person might wear several garments, such as socks, pants, shoes, etc. Some must be put on before the others. We need to **wear pants u before belt v**.



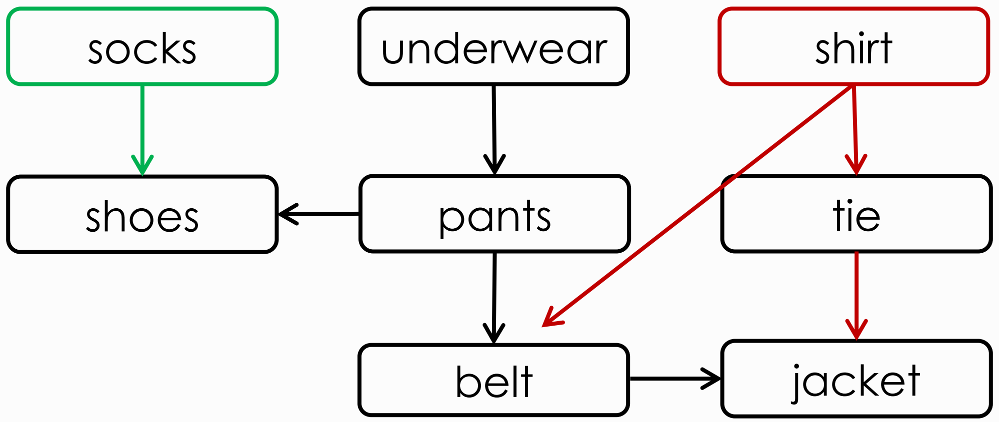
- A direct edge **`(u, v)`** in a DAG indicates that we need proceed **u** before **v** while respecting the dependency constraints. For example, we can put on garments in the following order.

underwear → shirt → socks → pants → **shoes** → belt → tie → jacket

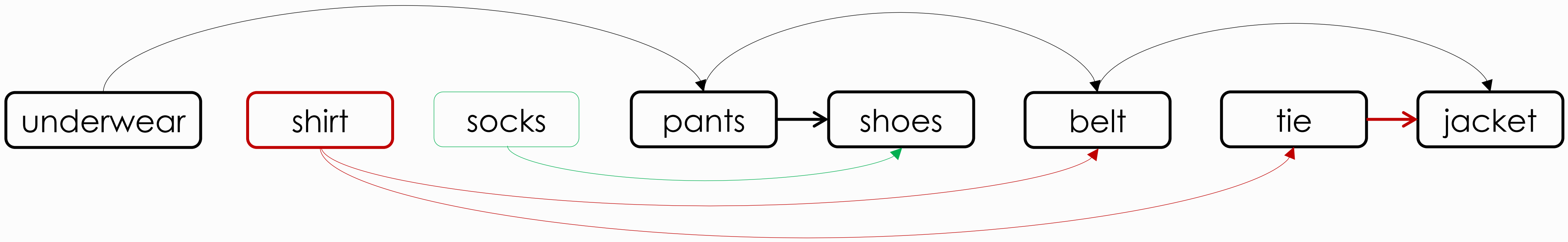
underwear → shirt → socks → pants → **belt** → **tile** → shoes → jacket

Topological Sort of DAG(위상 정렬)

- The following graph is "topologically unsorted graph".



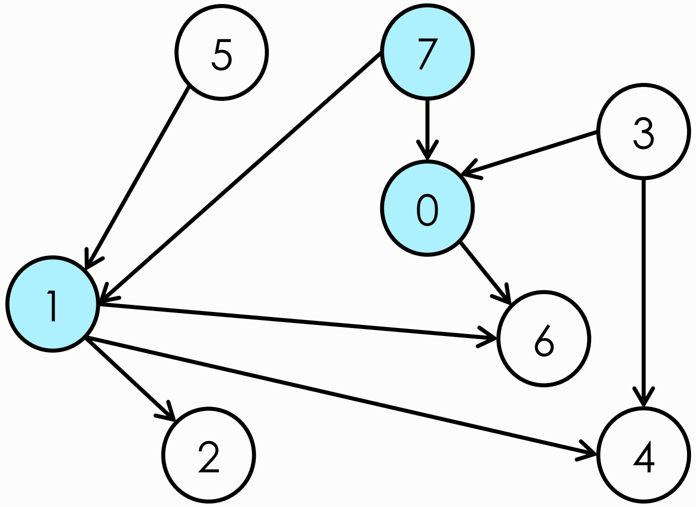
- The following graph is so called **topologically sorted graph**.



Topological Sort of DAG Example

1. Compute in-coming and out-going degrees of vertices 0, 1, and 7, respectively:
2. Select all vertices in a DAG that can be a starting vertex for the topological sort:

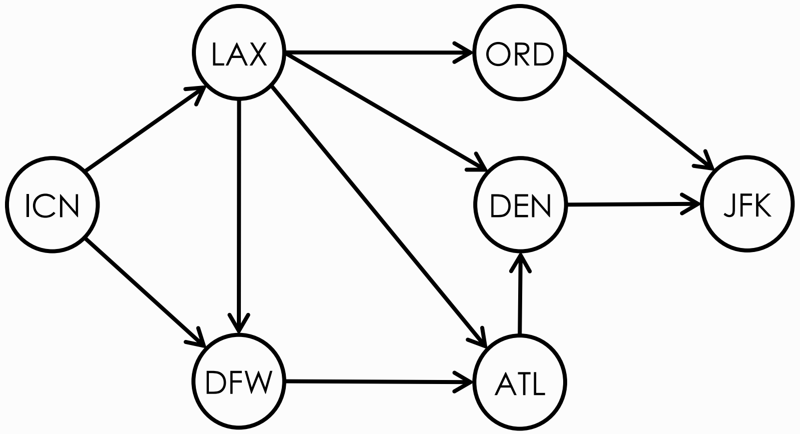
- **Observations:**
  - In a DAG, there must be at least one vertex that has in-coming degree of 0.
  - In a DAG, there can be many topological orderings shown below.
  - If there is a cycle in a DAG, a topological sort can not be defined since the precedence of vertices cannot be determined.




```
7   5   3   1   4   2   0   6
7   5   1   2   3   4   0   6
5   7   3   1   0   2   6   4
3   5   7   0   1   2   6   4
5   7   3   0   1   4   6   2
7   5   1   3   4   0   6   2
5   7   1   2   3   0   6   4
3   7   0   5   1   4   2   6
... more to come
```

Topological Sort - Kahn Algorithm

Idea: Keep on finding and removing vertices that have no incoming edges from the graph, one by one for all vertices.



Algorithm:

Compute indegree, **`ideg`**, for each vertex in **`g`**.
Put all vertices with 0 indegree into a **`queue.`**
Create an empty vertex list, **`path`**.
while **`queue:`**
  dequeue a vertex **`u`** from **`queue`**
  add **`u`** to tail of **`path`**
  for each vertex **`v`** in neighbor(**`u`**):
    decrease **`v`**'s indegree by 1
    if **`v`**'s indegree is 0:
      add **`v`** to **`queue`**
return **`path`**

toposort.txt
edge list

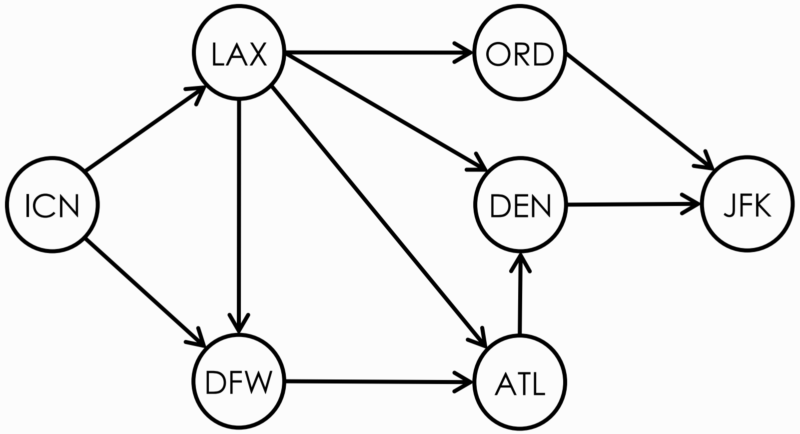
```
ATL DEN
ICN LAX
ICN DFW
DFW ATL
LAX DFW
LAX ATL
LAX ORD
LAX DEN
DEN JFK
ORD JFK
```

adjacency list

```
ATL: DEN
DEN: JFK
ICN: LAX DFW
LAX: DFW ATL ORD DEN
DFW: ATL
ORD: JFK
JFK:
```

Topological Sort - Kahn Algorithm

Idea: Keep on finding and removing vertices that have no incoming edges from the graph, one by one for all vertices.



Algorithm:

```
Compute indegree, ideg, for each vertex in g.
Put all vertices with 0 indegree into a queue.
Create an empty vertex list, path.
while queue:
    dequeue a vertex u from queue
    add u to tail of path
    for each vertex v in neighbor(u):
        decrease v's indegree by 1
        if v's indegree is 0:
            add v to queue
return path
```

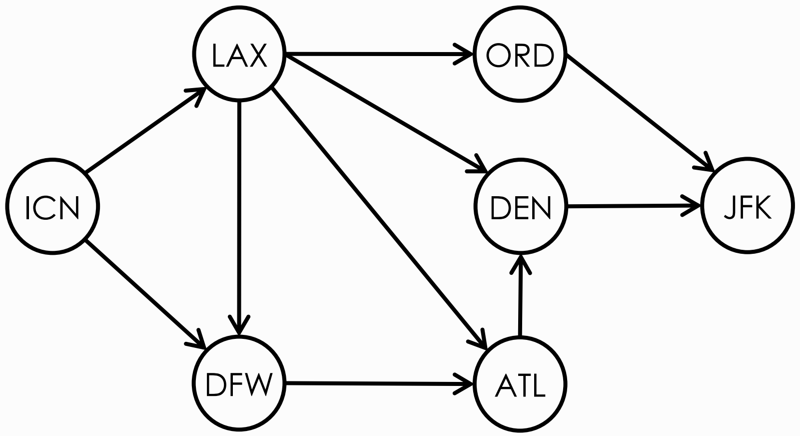
queue	*while **queue**: indegree at each vertex*
ICN	{ ATL:2, DEN:2, ICN:0, LAX:1, DFW:2, ORD:1, JFK:2 }
LAX	{ ATL:2, DEN:2, ICN:0, LAX:0, DFW:1, ORD:1, JFK:2 }
DFW, ORD	{ ATL:1, DEN:1, ICN:0, LAX:0, DFW:0, ORD:0, JFK:2 }
ORD, ATL	{ ATL:0, DEN:1, ICN:0, LAX:0, DFW:0, ORD:0, JFK:2 }
ATL	{ ATL:0, DEN:1, ICN:0, LAX:0, DFW:0, ORD:0, JFK:1 }
DEN	{ ATL:0, DEN:0, ICN:0, LAX:0, DFW:0, ORD:0, JFK:1 }
JFK	{ ATL:0, DEN:0, ICN:0, LAX:0, DFW:0, ORD:0, JFK:0 }

```
Topologically sort completed
['ICN', 'LAX', 'DFW', 'ORD', 'ATL', 'DEN', 'JFK']
```

Topological Sort - Kahn Algorithm Code

Algorithm:

Compute indegree, **ideg**, for each vertex in **g**.
Put all vertices with 0 indegree into a **queue.**
Create an empty vertex list, **path**.
while **queue:**
  dequeue a vertex **u** from **queue**
  add **u** to tail of **path**
  for each vertex **v** in neighbor(**u**):
    decrease **v**'s indegree by 1
    if **v**'s indegree is 0:
      add **v** to **queue**
return **path**




```
def topoSort(g):
    que = deque()
    ideg = dict()
    for v in g.vertices():
        ideg[v] = g.inDegree(v)
        if ideg[v] == 0:
            que.append(v)

    path = []
    while que:
        u = que.popleft()
        path.append(u)
        for v in g.neighbors(u):
            ideg[v] -= 1
            if ideg[v] == 0:
                que.append(v)
    return path

if __name__ == '__main__':
    g = Digraph('toposort.txt')
    print(topoSort(g))
```

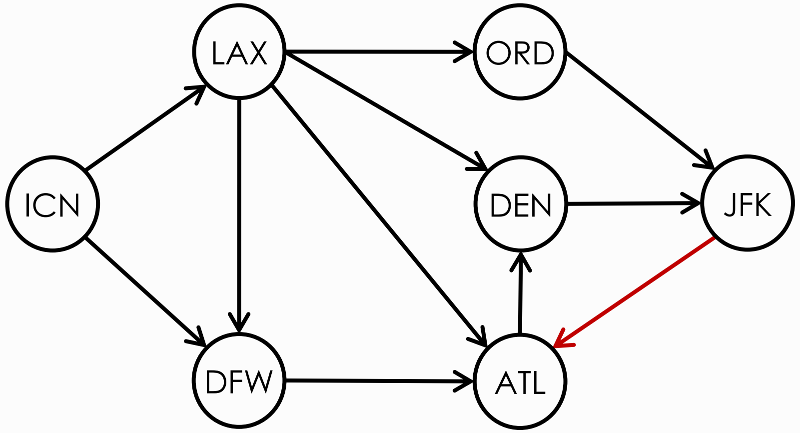
```
['ICN', 'LAX', 'DFW', 'ORD', 'ATL', 'DEN', 'JFK']
```

Topological Sort - Kahn Algorithm Coding Exercise

- Create a new file, topocycle.txt, by appending an edge from JFK to ATL to `toposort.txt`.
- Observe the result of the topological sort.
  - Print out the status of indegree of the graph.

```
if __name__ == '__main__':
    g = Digraph('topocycle.txt')
    print(topoSort(g))
```

```
['ICN', 'LAX', 'DFW', 'ORD']
```



*Create a new file, **topocycle.txt** by appending a new edge to toposort.txt*

- Enhance the code that detects a cycle such that it outputs the result shown below:

```
if __name__ == '__main__':
    g = Digraph('topocycle.txt')
    print(topoSort(g))
```

```
It is not a DAG - a cycle found.
None
```

  - Hint: Before returning the result of the topological sort, check the sum of indegree of vertices.

DFS with Recursion vs. Modified DFS for Topological sort

```
def dfs(g):
    path = []
    for u in g.vertices():
        if u not in path:
            _dfs(g, u, path)
    return path

def _dfs(g, u, path):
    path.append(u)
    for v in g.neighbors(u):
        if v not in path:
            _dfs(g, v, path)

g = Diraph('toposort.txt')
print(dfs(g))
```

```
def topoSort_dfs(g):
    path = []
    for u in g.vertices():
        if u not in path:
            _topoSort_dfs(g, u, path)
    return path

def _topoSort_dfs(g, u, path):
    for v in g.neighbors(u):
        if v not in path:
            _topoSort_dfs(g, v, path)
    path.insert(0, u)    # u must be in front of v in topological sort.

g = Diraph('toposort.txt')
print(topoSort_dfs(g))
```

```
['ATL', 'DEN', 'LAX', 'ICN', 'DFW', 'ORD', 'JFK']
```

```
['ICN', 'LAX', 'ORD', 'DFW', 'ATL', 'DEN', 'JFK']
```

- In standard DFS algorithm, it appends each vertex into the **path** immediately when visited. Since DFS can start at any vertex, it cannot guarantee to generate a topologically sorted list.
- Instead, we then put it to **the front of the result list**, **path** after all neighbors of a vertex **u** are visited. In this way, we can make sure that **u** appears before all its neighbors **v**'s in the sorted list.

Exercise: Modified DFS for Topological sort

```
def topoSort_dfs(g):
    path = []
    for u in g.vertices():
        if u not in path:
            _topoSort_dfs(g, u, path)
    return path

def _topoSort_dfs(g, u, path):
    for v in g.neighbors(u):
        if v not in path:
            _topoSort_dfs(g, v, path)
    path.insert(0, u)   # u must be in front of v in topological sort.

g = Diraph('toposort.txt')
print(topoSort_dfs(g))
```

```
It is not a DAG - a cycle found.
None
```

- What would happen if you run this code for a non-DAG such as `'topocycle.txt'`?
- Fix the code such that it detects a cycle if any and outputs as shown above.

Time Complexity

- The time complexity of DFS/BFS algorithms and Kahn’s topological sort algorithm are $\boldsymbol{O(V + E)}$, where V and E are the total number of vertices and edges in the graph, respectively.

Applications of Topological Sort

- Topological sorting is used mainly when tasks or items have to be ordered, where some tasks or items have to occur before others can.
  - **Scheduling jobs**, given dependencies some jobs have on some other jobs.
  - **Course arrangement** in educational institutions. Finding prerequisites of any job or task.
  - **Detecting deadlocks** in operating systems. Finding out if cycles exist in a graph.
  - Resolving symbol dependencies in linkers. Compile-time build dependencies. Deciding the appropriate order of performing compilation tasks in makefiles.

Summary

- **Digraph** class may be derived from **Graph** class and need to override some methods.
- For digraph traversals, we may use almost the same algorithms used for undirected graphs.

- A **DAG**(Directed acyclic graph) is a special case of **digraph**.
  A DAG will always have at least one valid topological sort possible.
  A DAG has at least one vertex that has no incoming edge.

- The graph must be a DAG for topological sort to be possible.
  If a graph is **undirected**, each edge being bidirectional creates a cycle between the two vertices it connects; hence topological sort is not possible.

학습 정리

1) 비순환 방향 그래프(DAG)는 노드들이 특정한 방향을 향하고 순환하는 구조가 없다

2) 위상(Topological) 정렬은 비순환 방향 그래프(DAG)에서 정점 (vertex)들을 선형으로 정렬하는 것이다

3) DAG가 아닌 그래프에서는 위상 정렬을 할 수 없다

4) 위상 정렬의 시간복잡도는 O(V + E)이다

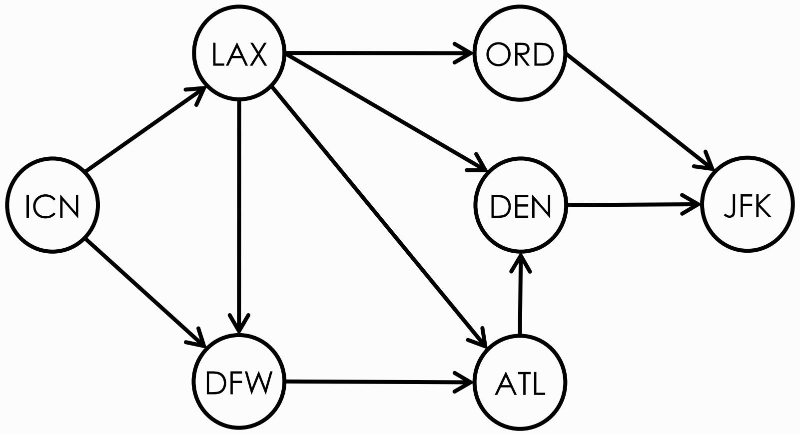

파이썬으로 배우는
데이터 구조
수고했습니다
곧 다음 시간에
다시 뵙겠습니
다

Topological Sort - Kahn Algorithm

Idea: Keep on finding and removing vertices that have no incoming edges from the graph, one by one for all vertices.

toposort.txt
edge list

```
ATL DEN
ICN LAX
ICN DFW
DFW ATL
LAX DFW
LAX ATL
LAX ORD
LAX DEN
DEN JFK
ORD JFK
```



adjacency list

```
ATL: DEN
DEN: JFK
ICN: LAX DFW
LAX: DFW ATL ORD DEN
DFW: ATL
ORD: JFK
JFK:
```