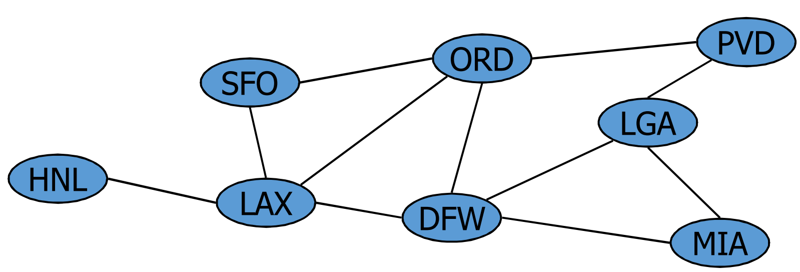
Chapter 3: Decomposition of graphs

3.1 Introduction to graph

3.1.1 Basic concepts

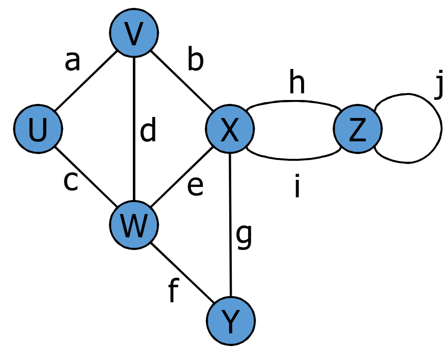
Graphs

- A graph G(***V, E***) consists a set vertices ***V*** and a set of edges ***E.***
- For instance, the graph below represents direct flight routes between some airports.



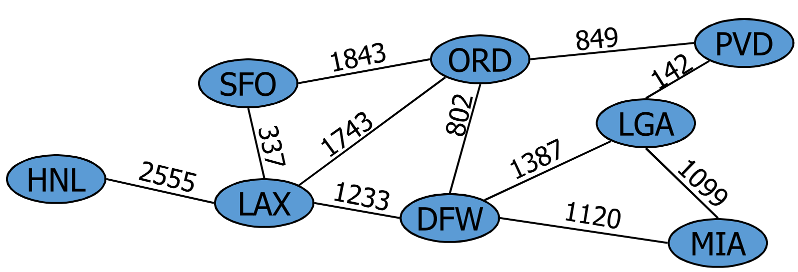
Some terminologies

- **Adjacent vertices:** U and V are adjacent
- **Edge incident(s) on a vertex:** a, d, and b are incidents on V
- **Degree of a vertex:** V's degree is 3
- **Parallel edges:** h and i are parallel edges
- **Self-loop:** j is a self-loop
- **Path:** V-U-W-Y is a path between V and Y
- **Cycle**: a path its starting and ending vertices are the same U-V-W-U
- **Connected**: if between any two vertices there is a path



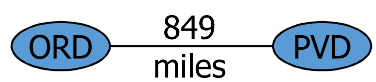
Weighted and unweighted graphs

- A graph is weighted if each of its edge is associated with a numeric weight. Otherwise, it is unweighted.



Directed and undirected graphs

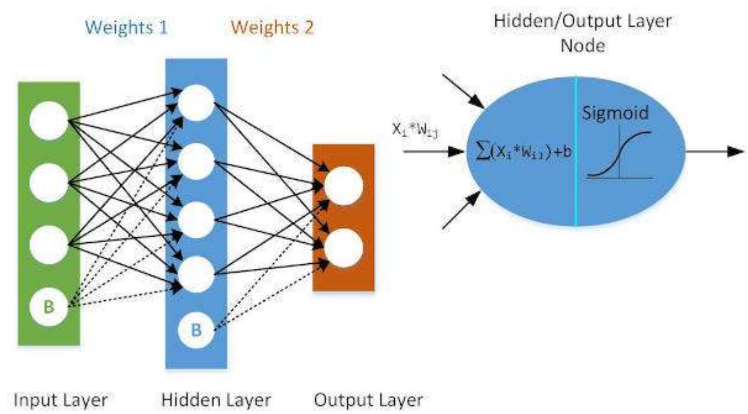
- A graph can be directed or undirected.
- For example: the flight AA 1206 is from ORD to PVD, and the distance between ORD and PVD is 849 miles.



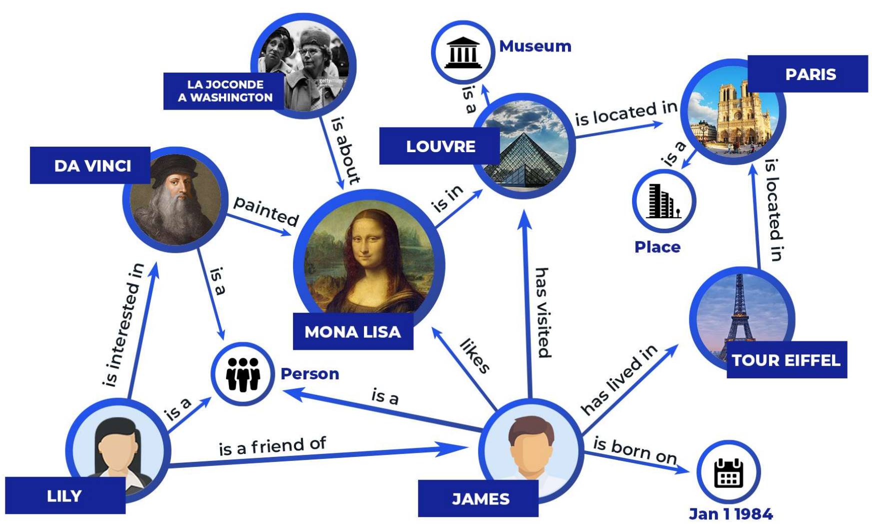
Broad applications of graph

- A wide range of problems can be expressed as graphs with clarity and precision. For instances,
  - Transportation networks, computer and communication networks, and social network
  - Databases entity-relationship diagram
  - Printed circuit board
  - Artificial neural network (ANN)
  - Knowledge graph
  - Probabilistic graphic model
  - etc.

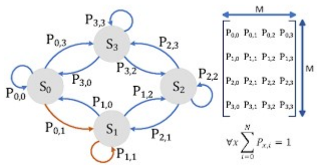
Artificial neural network

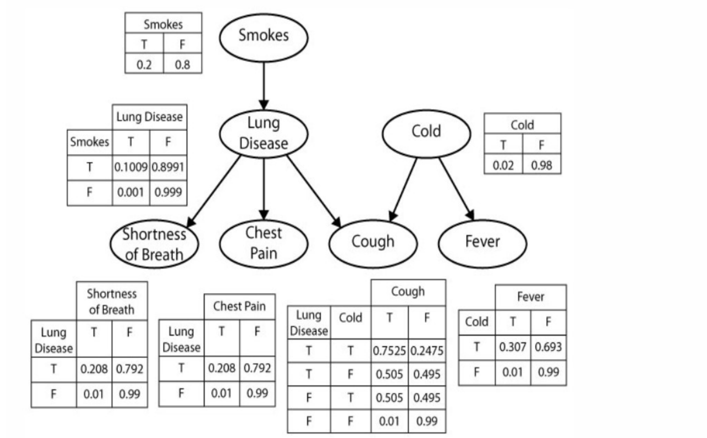


Knowledge graph



Markov chain and probabilistic graphic model

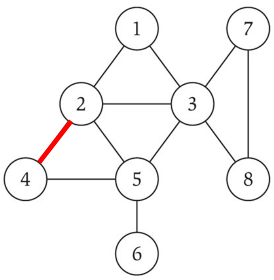




3.1.2 Graph representations

Graph adjacency matrix representation

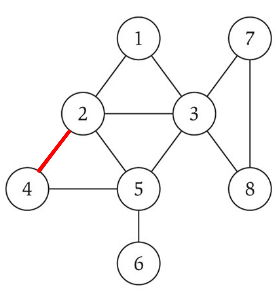
- A graph is a relation on the set of vertices.
- An n-vertex graph can be represented as an n x n adjacency matrix.



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Pros and cons of matrix representation

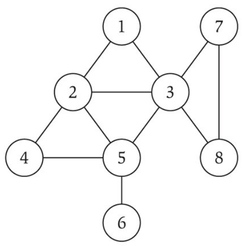
- Pros: Easy to understand; O(1) to locate an edge
- Cons: Spatial complexity O(n^2); Waste memory spaces for graphs sparsely connected.



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Adjacency list representation

- For each vertex, we associate it with its adjacency list.
- Pros. Reduced spatial complexity for an m-edge n-vertex graph from $O(n^2)$ to $O(m + n)$
- Cons. Checking if (u, v) is an edge takes O(deg(u)) rather than O(1) time.



1: 2 3
2: 1 3 4 5
3: 1 2 5 7 8
4: 2 5
5: 2 3 4 6
6: 5
7: 3 8
8: 3 7

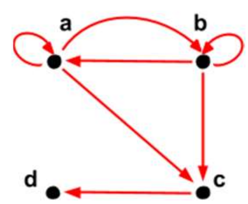
Highest degree of adj. list

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Example: representations of digraph

- Adjacency list representation:

```
graph = {'a': ['a', 'b', 'c'],
         'b': ['a', 'b', 'c'],
         'c': ['d'],
         'd': [ ]}
```



- Matrix representation:

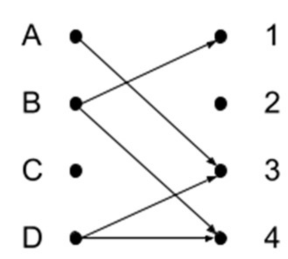
$$Graph = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Python built-in datatype `dict`

- Python built-in datatype dictionary `dict` is ready for graph.
- A dictionary in Python contains a collection of key-value pairs placed inside curled braces `{ },` separated by a comma.
- Each key-value pair in a Python dictionary specifies a relation, and is separated by a colon as `key : value`
- A dictionary looks like the follow:
  `d = {key1 : value1, key2 : value2 , key3 : value3 }`
- Let us implement a sample bipartite graph as an object of Python dictionary.

A sample bipartite graph as a Python dictionary

```
>>> g = {'A':3, 'B':{1, 4}, 'D':{3,4}}
>>> type(g)
<class 'dict'>
>>> len(g)
3
>>> g
{'A': 3, 'B': {1, 4}, 'D': {3, 4}}
>>> g.keys()
dict_keys(['A', 'B', 'D'])
>>> g.values()
dict_values([3, {1, 4}, {3, 4}])
```



Using key to retrieve a value from a dictionary

- One **cannot** use an index to retrieve a value of a dictionary. Instead, one should use key to retrieve a value as the follow:

```
dictionary_name[key]
```

- Example:

```
>>> for key in g:
   print(key, g[key])

A 3
B {1, 4}
D {3, 4}
```

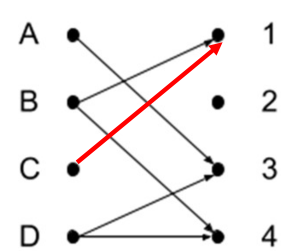
Adding a key-value pair a dictionary

- Dictionaries are mutable objects. You can add a new key-value pair to a dictionary with an assignment statement as:

```
dictionary_name[key] = value
```

- Example: adding the edge (C, 1) in the graph

```
>>> g['C'] = 1
>>> g
{'A': 3, 'B': {1, 4}, 'D': {3, 4}, 'C': 1}
>>>
```



Graph as an ADT (abstract data type)

- For a simple graph, one may hand code it easily as we just did.
- However, in general, we need an abstract data type (ADT).
- The script `graph_adj_dict.py` is an example.

```
class Vertex:
    def __init__(self, name):
        self.name = name
        self.neighbors = set()

    def add_neighbor(self, v):
        if v_name not in self.neighbors:
            self.neighbors = self.neighbors | {v}
```

builds adj. list!
the '|' is Union

Python implementation (cont.)

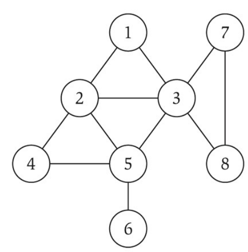
```
class Graph:
    graph = {}

    def add_vertex(self, vertex):
        if isinstance(vertex, Vertex) and \
           vertex.name not in self.graph:
            self.vertices[vertex.name] = vertex

# continued in the next page
```

Adds vertex to graph

Adjacency list: Python implementation

```
#continued from previous page
    def add_edge(self, u, v):
        if u in self.graph and v in self.graph:
            for key, value in self.graph.items():
                if key == u:
                    value.add_neighbor(v)
                if key == v:
                    value.add_neighbor(u)

    def print_graph(self):
        for key in sorted(list(self.vertices.keys())):
            print(key + str(self.vertices[key].neighbors))
```

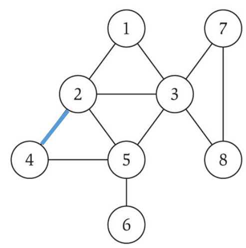
Test it with the sample graph




```
def main():
    g = Graph()
    for i in range(1, 9):
        g.add_vertex(Vertex(str(i)))
    edges = ['12', '13', '23', '24', '25', '35', \
             '37', '38', '45', '56', '78']
    for edge in edges:
        g.add_edge(edge[:1], edge[1:])
    g.print_graph()
main()
```

Hands on

- Run the test program `graph_adj_dict.py`;
- Visualize the execution, if you have question, through https://goo.gl/sQtVyl, and
- Compare the output with the graph in adjacency list



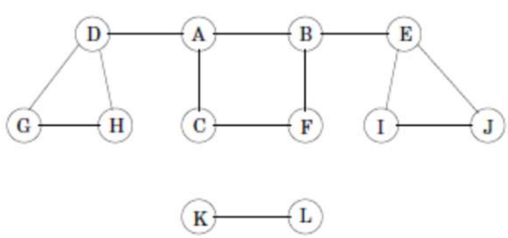
1	2	3			
2	1	3	4	5	
3	1	2	5	7	8
4	2	5			
5	2	3	4	6	
6	5				
7	3	8			
8	3	7			

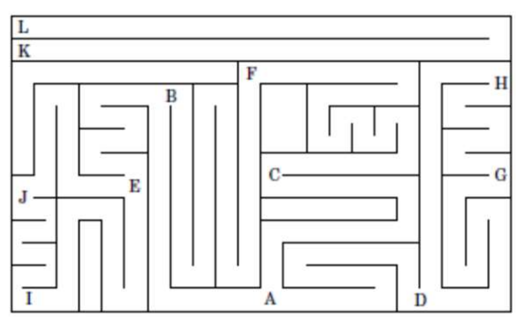
3.2 Depth-First Search (DFS) and applications

Graph traversal and reachability

- Graph traversal: a *systematic procedure* that *exploring a graph* by examining all of its vertices and edges.
- Graph traversal algorithms are key to answering many fundamental questions about graphs involving the notion of ***reachability***, that is, in determining how to travel from one vertex to another while following paths of a graph.

Exploring a graph is like navigating a maze





3.2.1 Depth First Search (DFS) algorithm

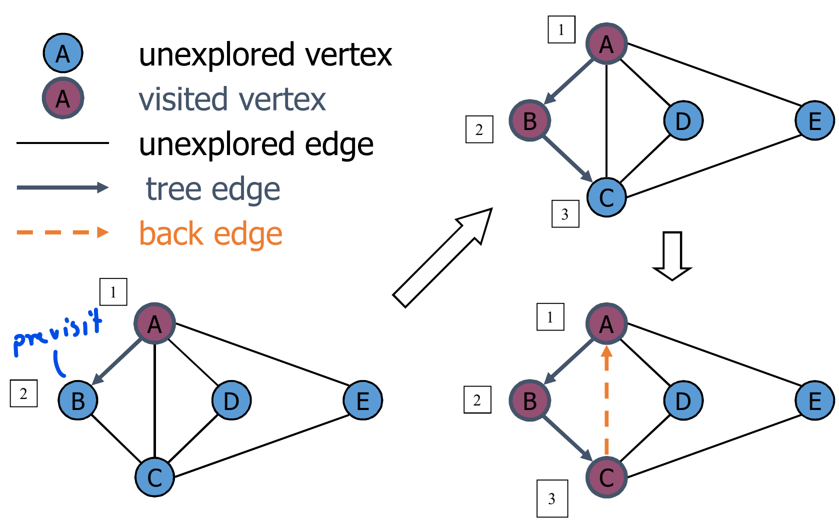
Depth first search

- Depth-first search (DFS) is a graph traversal algorithm.
- The algorithm starts at some arbitrarily selected node as the root node first
- Then it explores the graph as far as possible along each branch before backtracking.

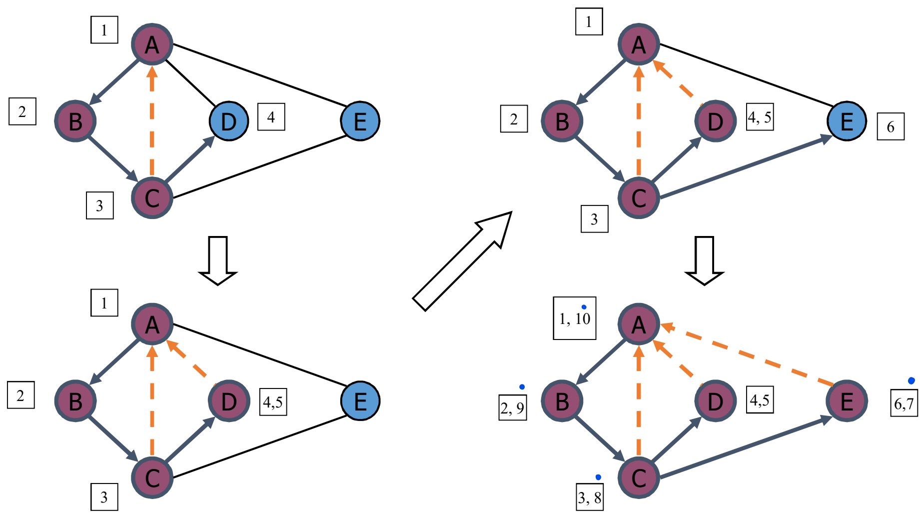
DFS from v to all nodes reachable in G

```
procedure explore(G, v)
Input:    G = (V, E) is a graph; v ∈ V
Output:   visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v) - label
for each edge (v, u) ∈ E:
   if not visited(u):  explore(u)
postvisit(v)
```

DFS illustrated: explore(G, A)



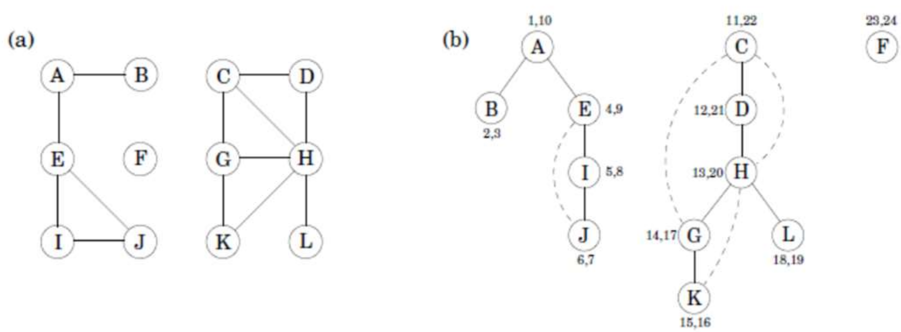
Example (cont.)



5 vertices
10 visits!

DFS for G with disjoint regions

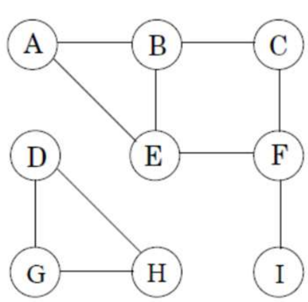
```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v):  explore(v)
```

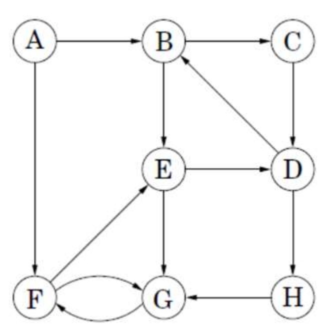


DFS in O(|V| + |E|)

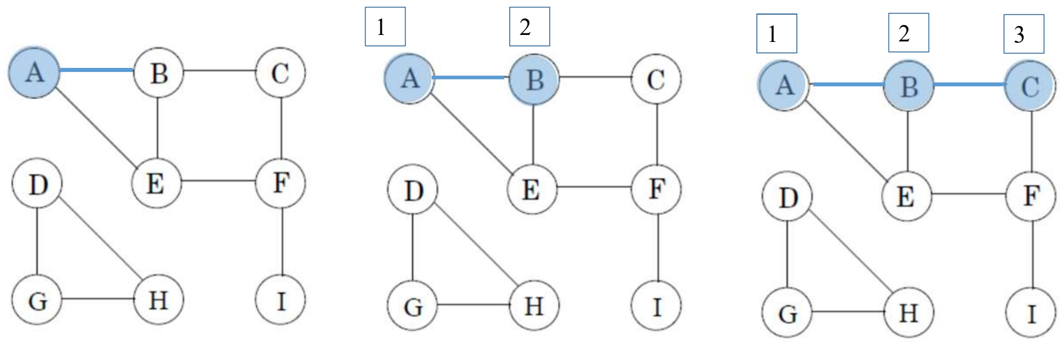
- Setting/getting a vertex/edge label takes ***O***(1) time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as TREE or BACK
- DFS runs in O(|V| + |E|) time provided the graph is represented by the adjacency list structure

DFS exercises: page.101: 3.1 and 3.2(a)

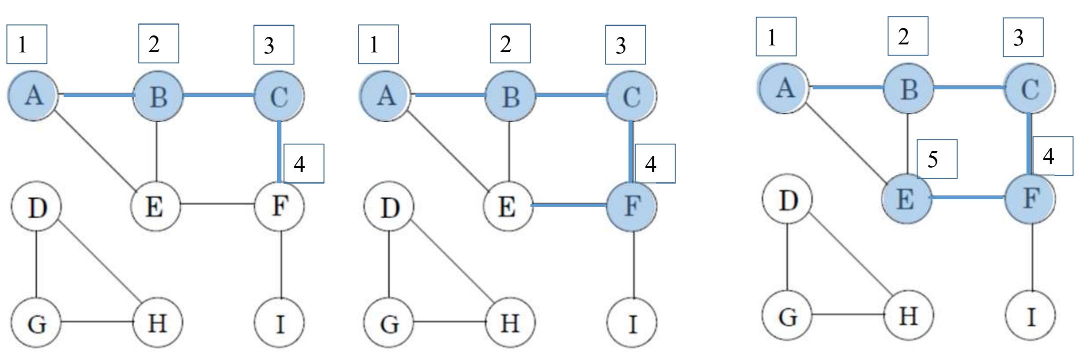




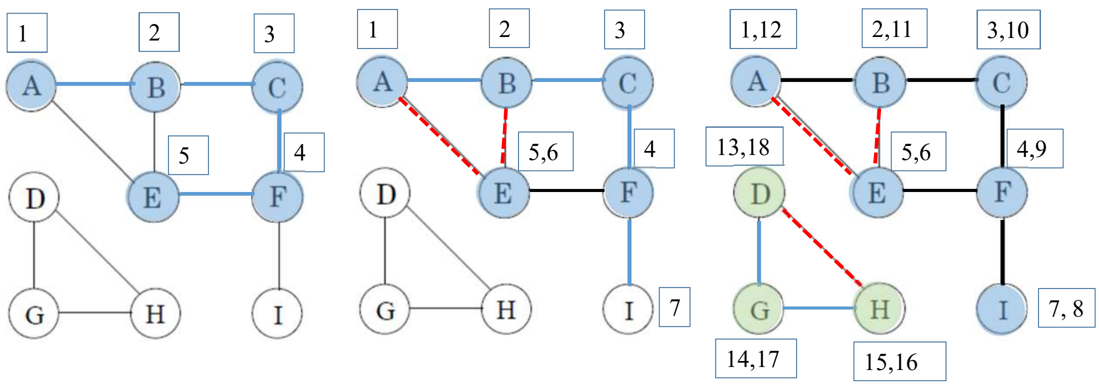
DFS exercise: 3.1 solution



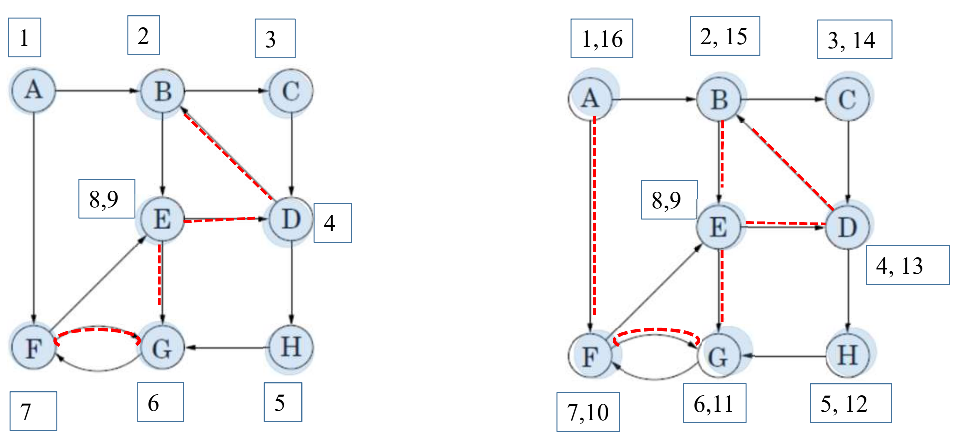
DFS exercises: page.101: 3.1



DFS exercise 3.1 solution (continue)



DFS exercise: 3.2(a) solution:



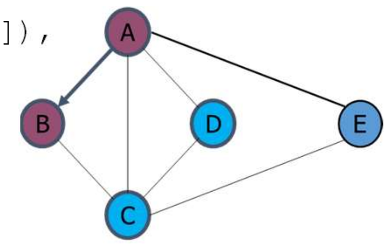
3.2.2 DFS in Python

Script: `dfs_recursive.py`

```
def dfs(graph, start, visited=None):
    global t
    t += 1
    print('DSF called ', t, 'times.')
    if visited is None:
        visited = set()
    visited.add(start)
    for key in graph[start] - visited:
        dfs(graph, key, visited)
    return visited
```

Apply it

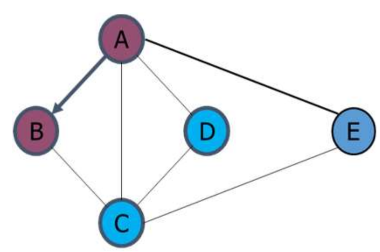
```
graph = {'A': set(['B', 'C', 'D', 'E']),
         'B': set(['A', 'C']),
         'C': set(['A', 'B', 'D', 'E']),
         'D': set(['A', 'C']),
         'E': set(['A', 'C'])}

t = 0
v = dfs(graph, 'A')
```



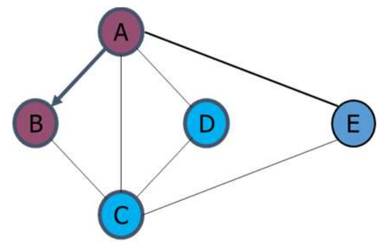
You may visualize step-by-step execution via https://goo.gl/XZ25DT

An iterative implementation

- With Python ADT set, we can implement the DFS algorithm iteratively in only few lines too.
- See `dfs_iterative.py` next page



Code: `dfs_iterative.py`




```
# http://eddmann.com/posts/depth-first-search...
def _dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        #print('The stack is:', stack)
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
            #print('The visited is: ', visited)
    return visited
V = _dfs(graph, 'A')
```

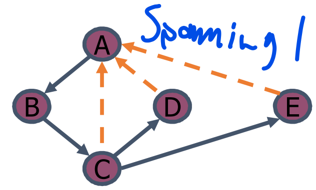
To visualize the execution visit https://goo.gl/3P6F6x

3.2.3 DFS applications

Solve graph problems with DFS

- Depth-first search (DFS) is a general technique for traversing a graph. Applying it, you solve some interesting problems.
- For example:
  - Find a spanning tree of connected vertices in a graph: the tree edges
  - Find connectivity of a graph: checking if |V| = |visited| or not
  - Find a path between two given vertices: DFS from the start node and stop when reached the end node
  - Find a cycle in the graph: accepting a back edge.

Spanning tree and forest



Spanning

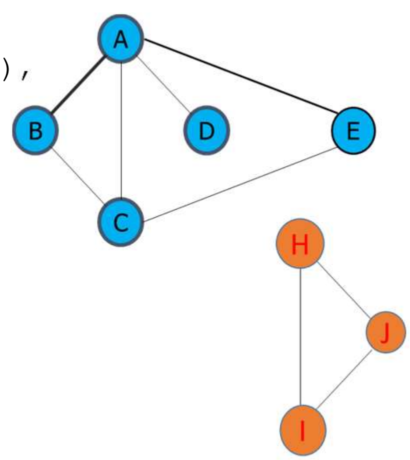
- A spanning tree of a connected component of graph G(v, e) contains all vertices of that component without a cycle.
- DFS on a connected graph results in a spanning tree.
- Spanning trees for multiple connected components of a graph form a spanning forest of a graph.

Finding number of connected components

- Question: How do you apply DFS to find the number of connected components of a graph? spanning forest
- Hints:
  1. A set of all vertices in a graph: V
  2. Component = { }
  3. While V is not empty:
     i. for a v in V, perform dfs with v as starting vertex, to obtain the visited vertex set V'
     ii. Component += V'
     iii. V = V – V'
  4. Number of components = len(Component)

Code: `dfs_iterative.py`

```
graph = {'A': set(['B', 'C', 'D', 'E']),
         'B': set(['A', 'C']),
         'C': set(['A', 'B','E']),
         'D': set(['A']),
         'E': set(['A', 'C']),
         'H': set(['I', 'J']),
         'I': set(['H', 'J']),
         'J': set(['H', 'I'])
}

t = 0
v = _dfs(graph, 'A')
```



Let us run `dfs_iterative.py` with sample graph2 to start with A, and then pick a vertex not visited.

gives a path, not necessarily in order.

Path finding

- We can specialize the DFS algorithm to find a path between two given vertices ***u*** and ***z*** using the template method pattern
- We call ***DFS***(***G, u***) with ***u*** as the start vertex
- We use a stack ***S*** to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex ***z*** is encountered, we return the path as the contents of the stack

Code: `dfs_path.py` (iterative)

```
def dfs_path(graph, start, goal):
    stack = [(start, [start])]
    visited = set()
    while stack:
        (vertex, path) = stack.pop()
        if vertex not in visited:
            if vertex == goal:
                return path
            visited.add(vertex)
            for neighbor in graph[vertex]:
                stack.append((neighbor, path + \
                              [neighbor]))
```

Test it

- Here is a simple test graph

```
graph = {'A': set(['B', 'E']),
         'B': set(['A', 'C']),
         'C': set(['B', 'D', 'E']),
         'D': set(['C']),
         'E': set(['A', 'C'])}
```

- Test it:

```
v = dfs_path(graph, 'A', 'D')
print (v)
```

- Visualize the execution through: https://goo.gl/ICSDWi

Comments

- The `dfs_path` function finds a path for a connected graph.
- If there is no path between two vertices, then the graph is not completely connected.
- Can we find ***all*** paths between two vertices in a connected graph?
- Yes, here is another implementation with Python.
- However, we need to know more about Python to apply yield and yield from. It is beyond the scope of this course. You may want to learn it through online search.

Code: `dfs_paths.py`

```
def dfs_paths(graph, start, goal):
    stack = [(start, [start])]
    while stack:
        (vertex, path) = stack.pop()
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                stack.append((next, path + [next]))
```

Test it

- Here is the simple test graph

```
graph = {'A': set(['B', 'E']),
         'B': set(['A', 'C']),
         'C': set(['B', 'D', 'E']),
         'D': set(['C']),
         'E': set(['A', 'C'])}

v = dfs_paths(graph, 'A', 'D')
print (v)
```

- Test it and visualize it: https://goo.gl/93TRg9

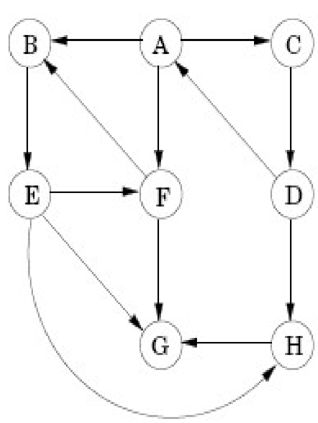
Cycle finding

- A cycle starts and ends at the same vertex.
- Applying the DFS algorithm to find a simple cycle, you keep track of the path between the start vertex and the current vertex.
- As soon as a back edge (***v, w***) is encountered, we return the cycle as the portion of the stack from the top to vertex ***w***
- However, a simple implementation may result in infinite loop. We study cycle in digraphs in the next section.

3.3 Digraphs

3.3.1 DFS in digraphs

DFS in digraphs



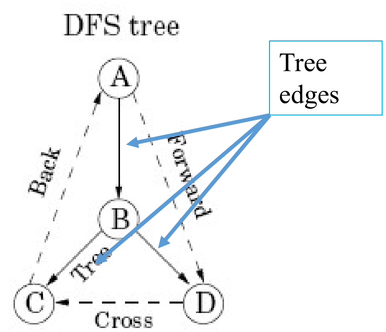
- The DFS algorithm can be applied to digraphs by following only the direction of each edge.

```
graph= {'A': set(['B', 'C', 'F']),
        'B': set(['E']),
        'C': set(['D']),
        'D': set(['A', 'H']),
        'E': set(['F', 'G', 'H']),
        'F': set(['B', 'G']),
        'G': set(),
        'H': set(['G'])}
```

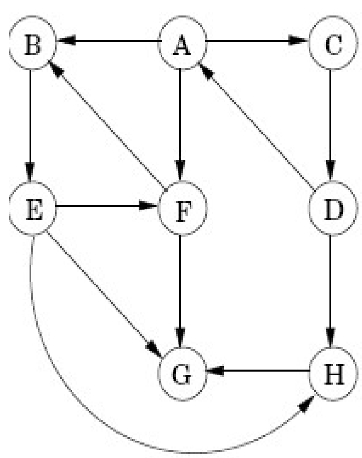
https://goo.gl/06q8bU

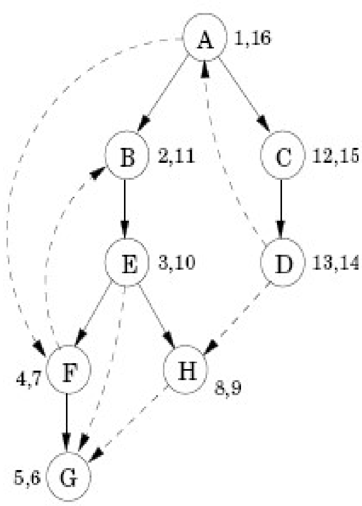
Tree, forward, back, and cross edges

- **Tree edges**: part of the DFS forest.
- **Forward edges:** lead from a node to a non-child descendant
- **Back edges:** lead to an ancestor in the DFS tree.
- **Cross edges:** lead to neither descendant nor ancestor; they lead to a node that has already been completely explored

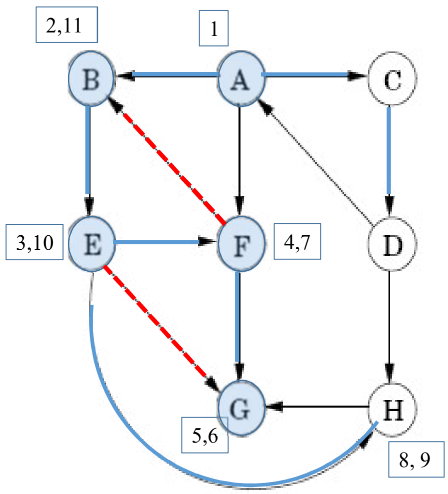


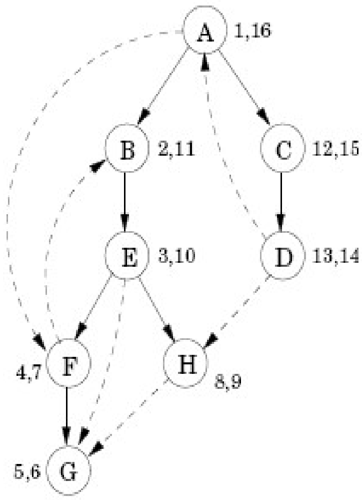
Example: DFS starting from A





Example: DFS starting from A

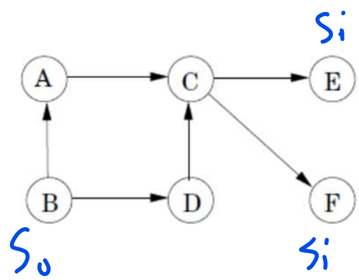


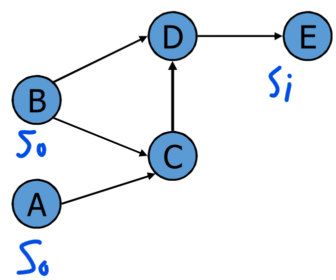


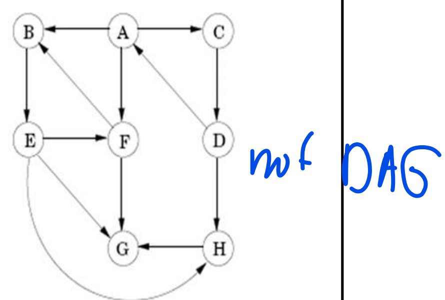
3.3.2 DAG: directed acyclic graph

DAG: directed acyclic graph

- A directed acyclic graph (DAG) is a digraph without a cycle.
- The vertex in a dag without an incoming edge is called a source.
- The vertex in a dag without an outgoing edge is called a sink.
- A dag may have multiple sources and sinks.
- Quiz: Which graph below is a dag? If it is, determine its source(s) and sink(s).

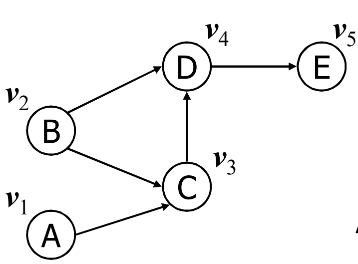


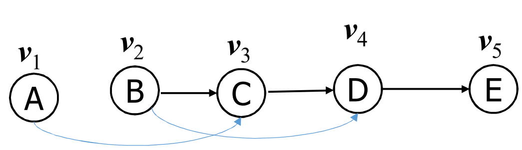




Topological order of a DAG

- A *topological ordering* of a dag is a numbering $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ of the vertices such that for every edge $(\boldsymbol{v}_i, \boldsymbol{v}_j)$, we have $\boldsymbol{i} < \boldsymbol{j}$. Finding a topological order of a dag is called *topological sorting.*
- **Theorem**: A digraph has a topological ordering if and only if it is a DAG.



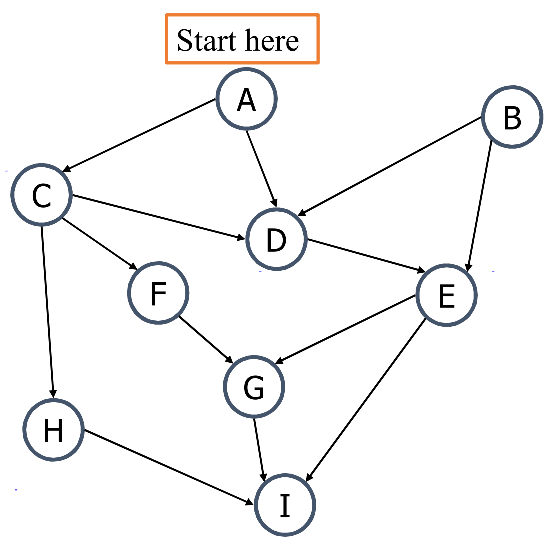


A linearization of the dag in its topologic order.

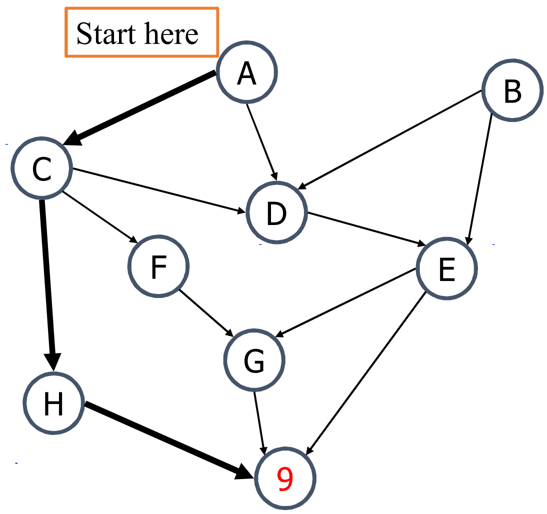
Topological sorting with DFS

- By applying depth first search, we can sort a DAG topologically:
  - Starting from a source (a vertex without incoming edge)
  - DFS to nowhere to go, then label the last node and cast all edges adjacent to that vertex
  - Back up
- We may identify a source node in a dag through observation.
- Or, reversing all edges first, then starting from any node for DFS to find a sink. A sink in a reversal digraph is a source in the original graph.
- See an example next page

Example:

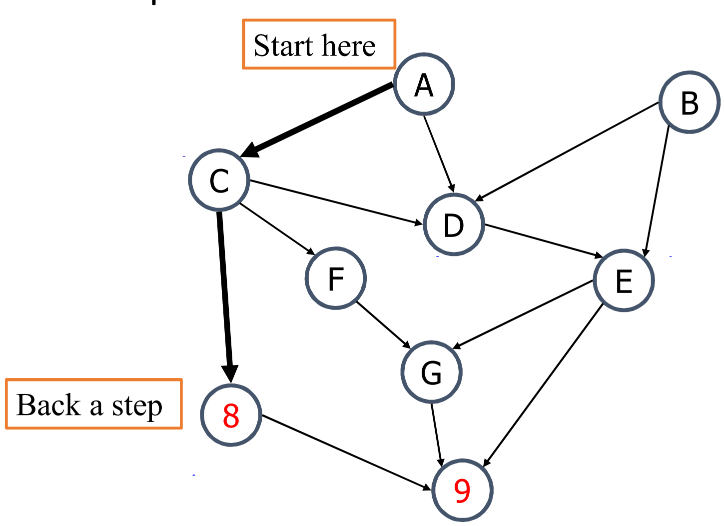


Example



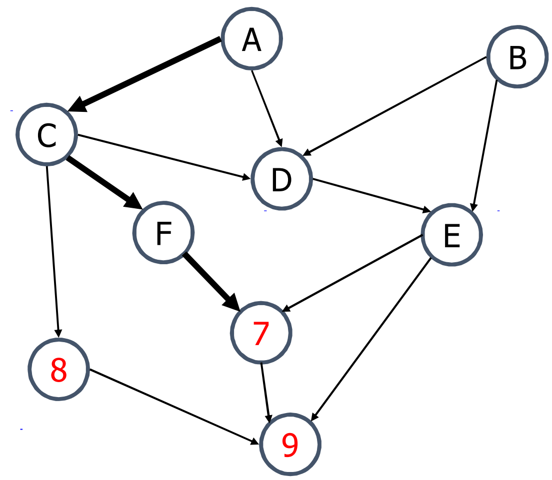
I: 9

Example



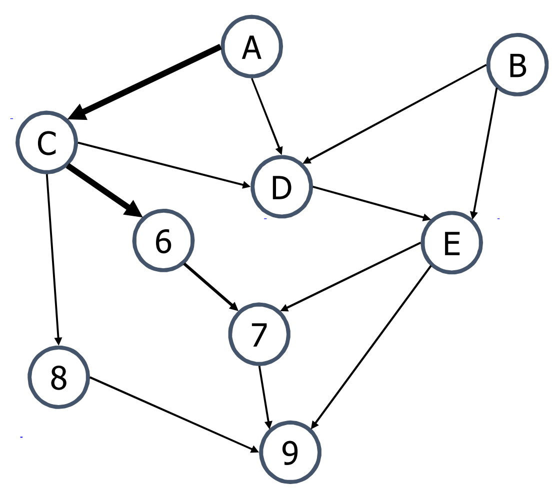
H: 8
I: 9

Example



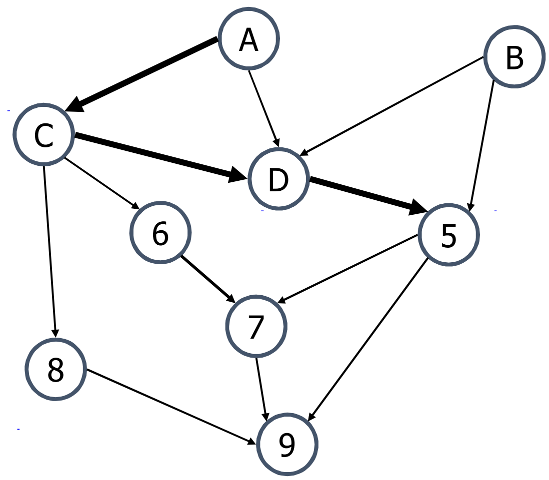
G: 7
H: 8
I: 9

Example



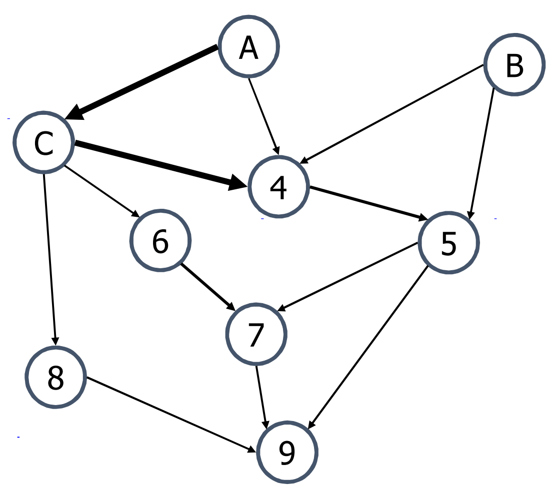
F: 6
G: 7
H: 8
I: 9

Example



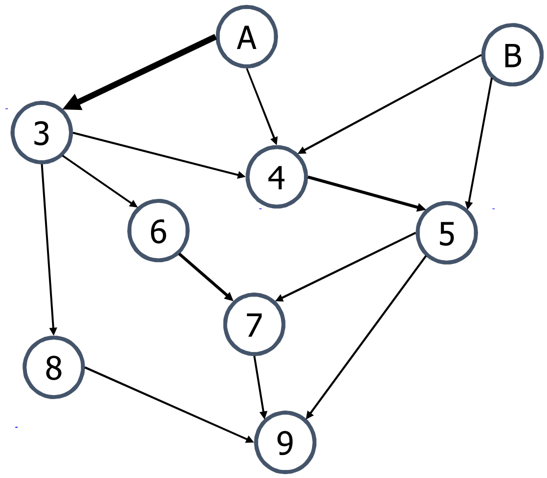
E: 5
F: 6
G: 7
H: 8
I: 9

Example



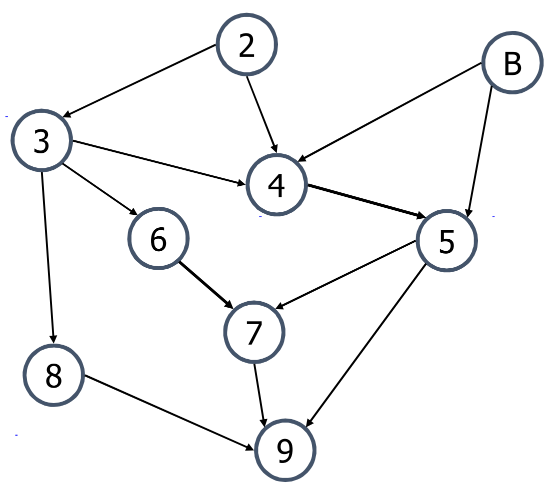
D: 4
E: 5
F: 6
G: 7
H: 8
I: 9

Example

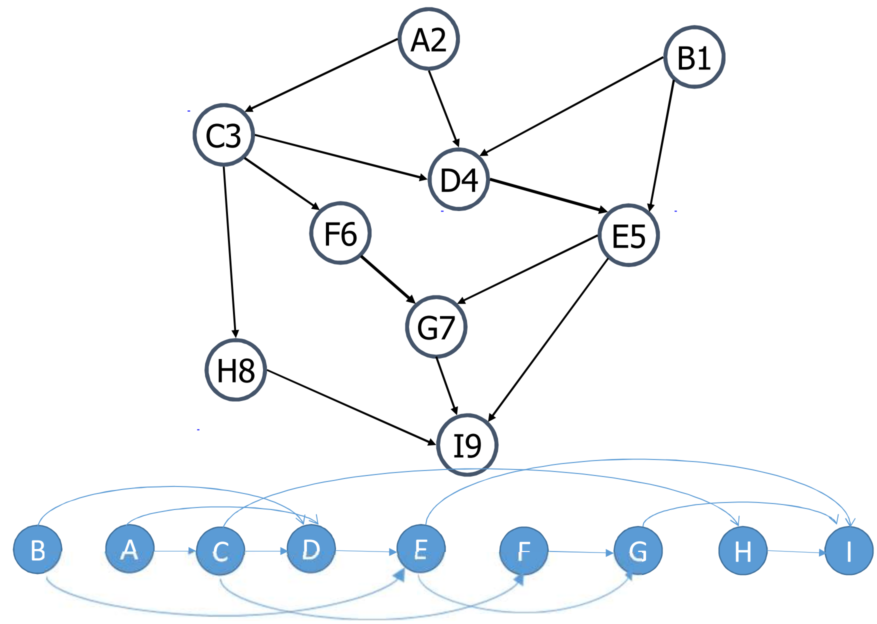


C: 3
D: 4
E: 5
F: 6
G: 7
H: 8
I: 9

Example



A: 2
C: 3
D: 4
E: 5
F: 6
G: 7
H: 8
I: 9



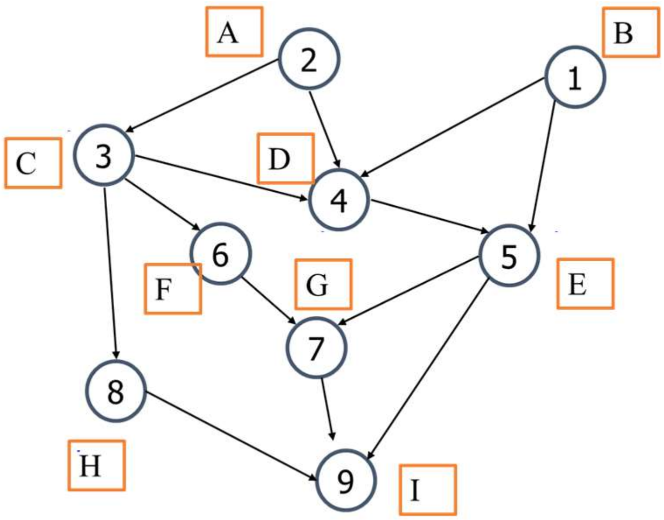
B: 1
A: 2
C: 3
D: 4
E: 5
F: 6
G: 7
H: 8
I: 9

Code: `dfs_tpl_order.py`

```
def dfs_tpl_order(graph,start,path):
     global n
     path = path + [start]
     for edge in graph[start]:
        if edge not in path:
            path = dfs_tpl_order(graph, edge, path)
    print (n, start)
    n -= 1
    return path
```

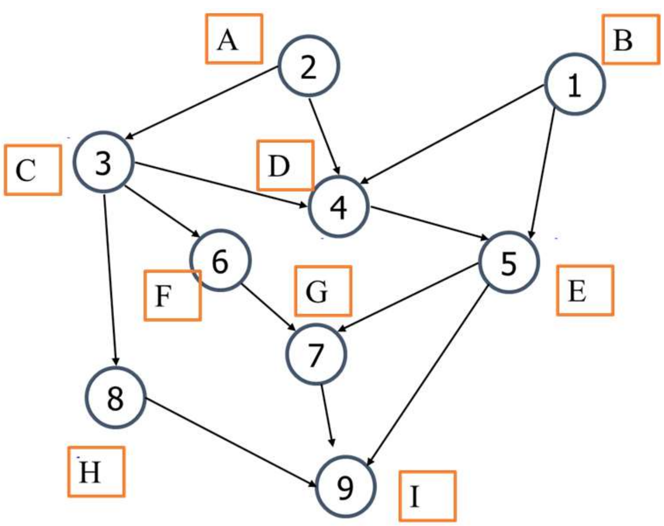
https://goo.gl/13cFu2

Test graph



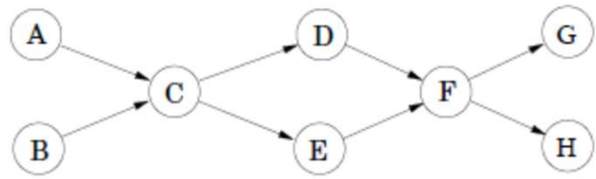

```
graph = {'A': set(['C', 'D']),
         'B': set(['D', 'E']),
         'C': set(['D', 'F', 'H']),
         'D': set(['E']),
         'E': set(['G', 'I']),
         'F': set(['G']),
         'G': set(['I']),
         'H': set(['I']),
         'I': set()}
```

Hands on:

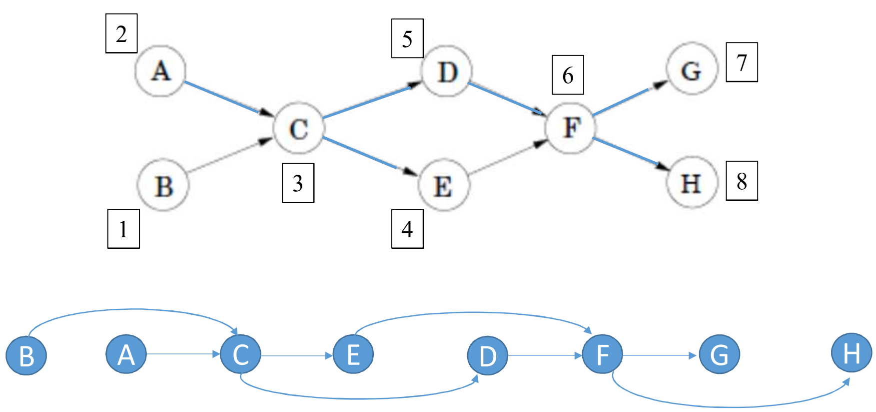


- Run `digraph_tpl_order.py`
- Sketch the linearized graph
- Notes: Starting from a different source, the result may not be the same.
- Even from the same source node, the result in each test run *may not be exactly the same*.
- One cannot determine an order if a digraph contains a cycle.

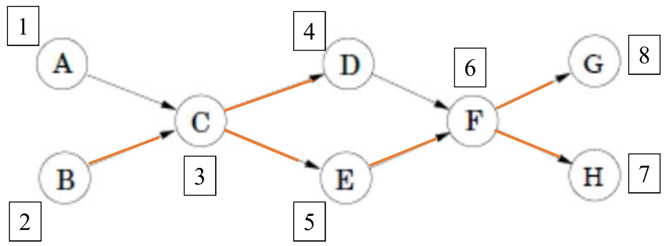
In class exercise: linearize the DAG below

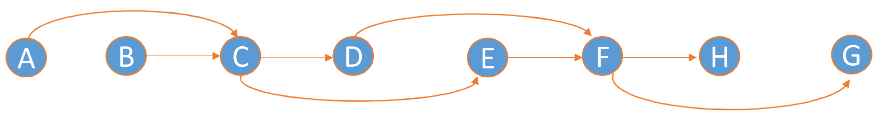


One solution

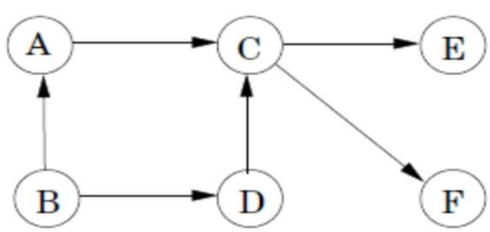


Another solution

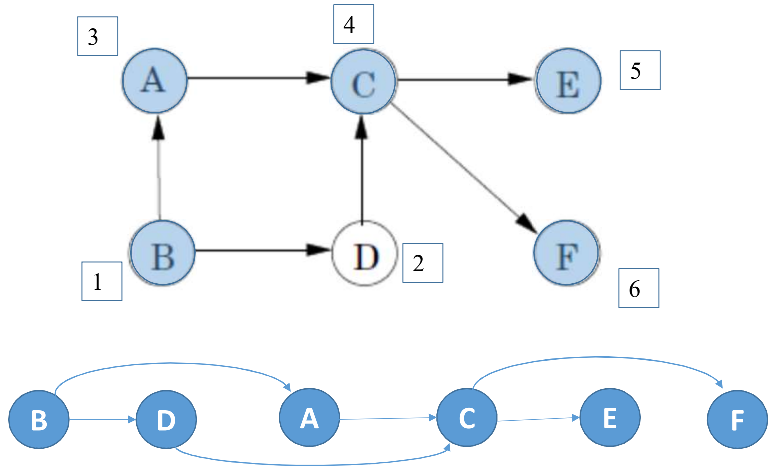




In class exercise: linearize the DAG below

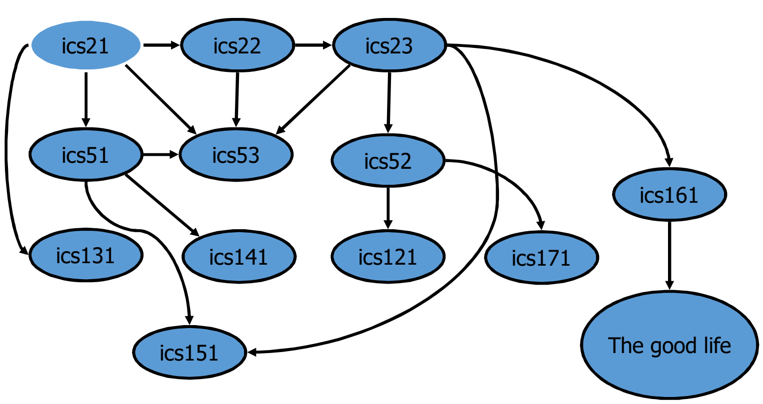


One solution



Scheduling: an application of DAG

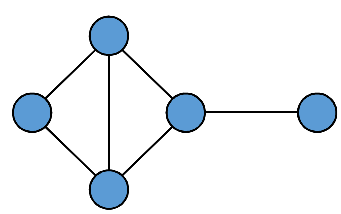
- The digraph illustrate the prerequisites of the ICS courses.
- Hands on: Linearizing it for establishing a plan of study.



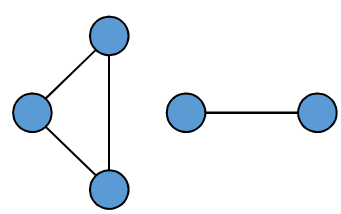
3.4 Strongly connected components

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G



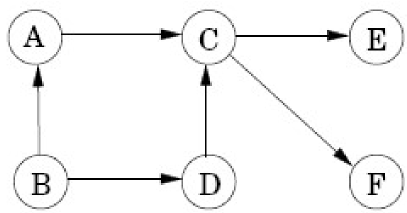
Connected graph



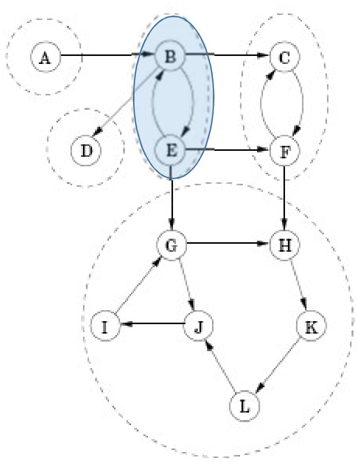
Non connected graph with two connected components

Strong connectivity for directed graph

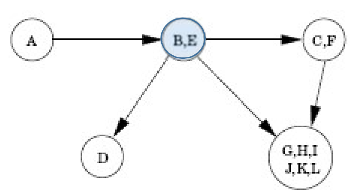
- A digraph ***G*** is strongly connected if there is a path from *u* to *v* and a path from *v* to *u* for any two nodes *u* and *v* in G.
- The digraph below is NOT strongly connected! In fact, all DAGs are not strongly connected.



Strongly connected components of a digraph

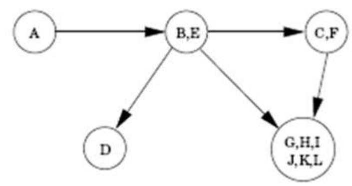



```
graph = {'A': set(['B']),
        'B': set(['D', 'C', 'E']),
        'C': set(['F']),
        'D': set(),
        'E': set(['B', 'F', 'G']),
        'F': set(['C','H']),
        'G': set(['H', 'J']),
        'H': set(['K']),
        'I': set('G'),
        'J': set(['I']),
        'K': set(['L']),
        'L': set(['J'])        }
```



Digraph decomposition

- *Every digraph is a dag of its strongly connected components.*
- Decomposing a digraph into strongly connected components is very informative and useful.
- An algorithmic approach is to
  - Find a strongly connected component containing a sink in the meta-graph
  - Remove it and repeat

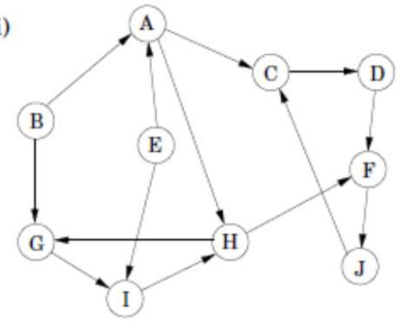


Hands on:

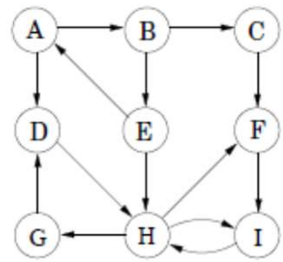
- Run the scripts `scc.py`
- Linearize the sample graph as a dag of connected components of the graph.

Exercises: strongly connected component

(i)



(ii)



Summary

- Graph representation
- DFS
- Applications of DFS: spanning tree/forest, path, paths, cycle
- DFS on digraphs: DAG, topological sorting, strong connectivity
- Decompose a digraph as its strongly connected components.

Free graph packages in Python

- NetworkX https://networkx.github.io/
- SNAP.py https://snap.stanford.edu/snappy/
- Python-graph: https://pypi.python.org/pypi/python-graph