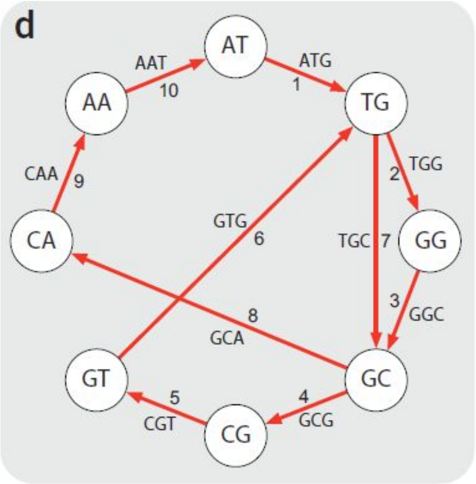
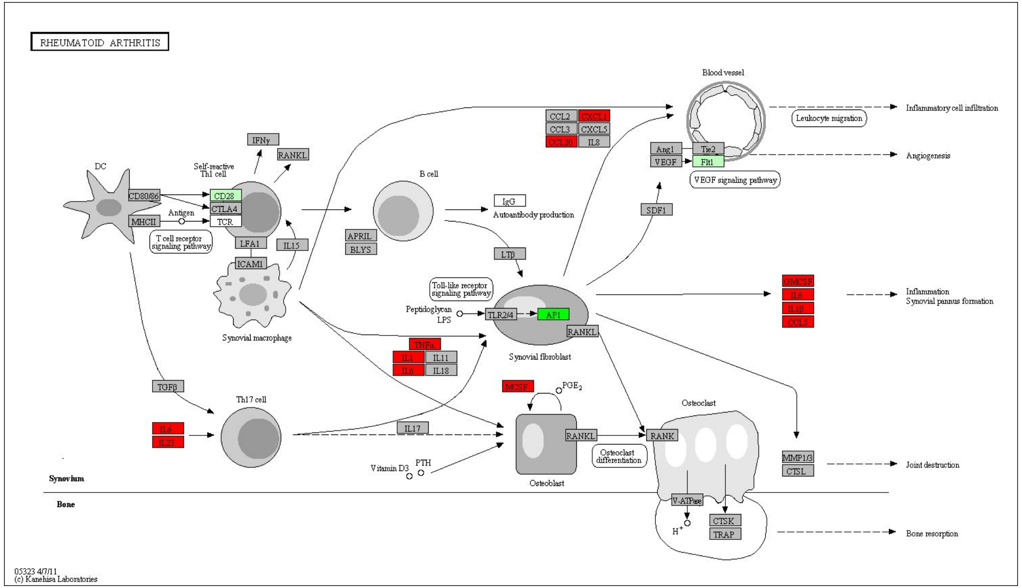
Scientific Programming: Part B

Graphs

Luca Bianco - Academic Year 2019-20
luca.bianco@fmach.it
[credits: thanks to Prof. Alberto Montresor]

Graphs: examples

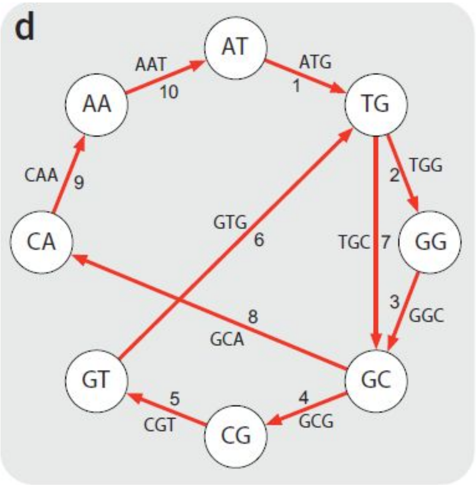


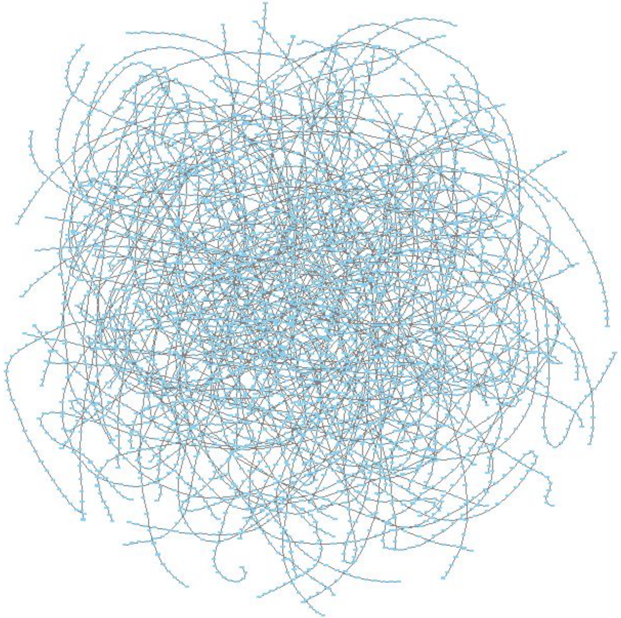


[From: Compeau et al, How to apply de Bruijn graphs to genome assembly, Nature Biotech,2011]

http://www.kegg.jp/

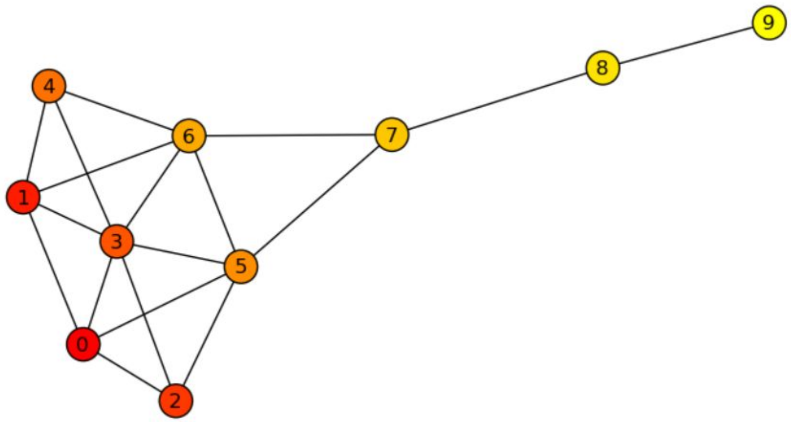
Graphs: examples





[From: Compeau et al, How to apply de Bruijn graphs to genome assembly, Nature Biotech,2011]

Graphs: examples



A 10 actor social network introduced by David Krackhardt to illustrate: degree, betweenness, centrality, closeness, etc. The traditional labeling is:
Andre=1, Beverley=2, Carol=3, Diane=4,
Ed=5, Fernando=6, Garth=7, Heather=8, Ike=9,
Jane=10.
[Social Network analysis for startups, "O'Reilly Media, Inc.", 2011]

The London underground system

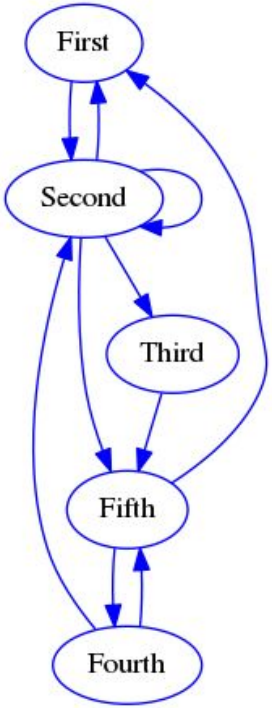
Graphs

Graph:

$G = (V,E)$

Where V and E are finite sets:

- V is the set of **nodes** (i.e. ‘things’)
- E is the set of **edges** (i.e. relationships among things) $E : V \times V$

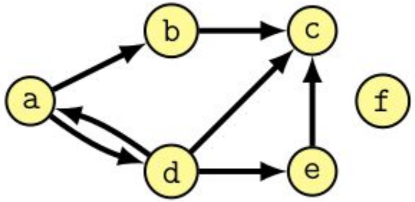


Graphs

Directed graph $G = (V, E)$

- V is a set of vertexes/nodes
- E is a set of edges, i.e. ordered pairs (u, v) of nodes

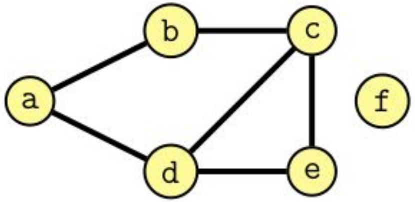
```
V = { a,b,c,d,e,f }
E = { (a,b),(a,d),(b,c),(d,a)
      (d,c),(d,e),(e,c) }
```



Undirected graph $G = (V, E)$

- V is a set of vertexes/nodes
- E is a set of edges, i.e. unordered pairs $[u, v]$ of nodes

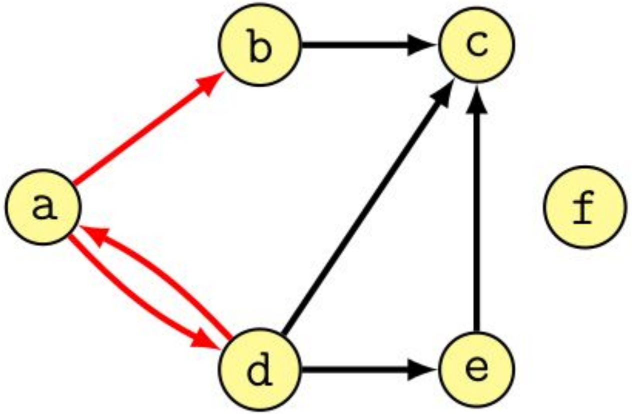
```
V = { a,b,c,d,e,f }
E = { [a,b],[a,d],[b,c],
      [c,d],[d,e],[c,e] }
```



Relations represented by edges can be **symmetric** (e.g. sibling_of: if X is sibling of Y then Y is sibling of X) and in this case the edges are just lines rather than arrows. In this case the graph is **directed**. In case relationships are not symmetric (i.e. $X \rightarrow Y$ does not imply $Y \rightarrow X$) we put an arrow to indicate the direction of the relationship among the nodes and in this case we say the graph is **undirected**.

Definitions

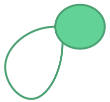
- Vertex v is adjacent to u if and only if $(u, v) \in E$.
- In an undirected graph, the adjacency relation is symmetric
- An edge (u, v) is said to be incident from u to v



- (a, b) is incident from a to b
- (a, d) is incident from a to d
- (d, a) is incident from d to a
- b is adjacent to a
- d is adjacent to a
- a is adjacent to d

Size and complexity

Ignoring self loops



Definitions

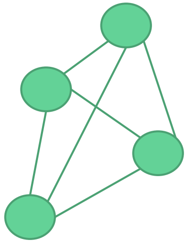
- $n = |V|$: number of nodes
- $m = |E|$: number of edges

Relationships between n and m

- In an undirected graph, $m \leq \frac{n(n-1)}{2} = O(n^2)$
- In a directed graph, $m \leq n^2 - n = O(n^2)$

Complexity of graph algorithms

- The computational complexity is measured based on both n and m (e.g. $O(n+m)$)



Undirected graph
n= 4
m = 6 (=4*3/2)

Size and complexity

Definitions

- $n = |V|$: number of nodes
- $m = |E|$: number of edges

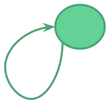
Relationships between n and m

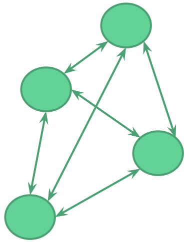
- In an undirected graph, $m \leq \frac{n(n-1)}{2} = O(n^2)$
- In a directed graph, $m \leq n^2 - n = O(n^2)$

Complexity of graph algorithms

- The computational complexity is measured based on both n and m (e.g. $O(n+m)$)

Ignoring self loops

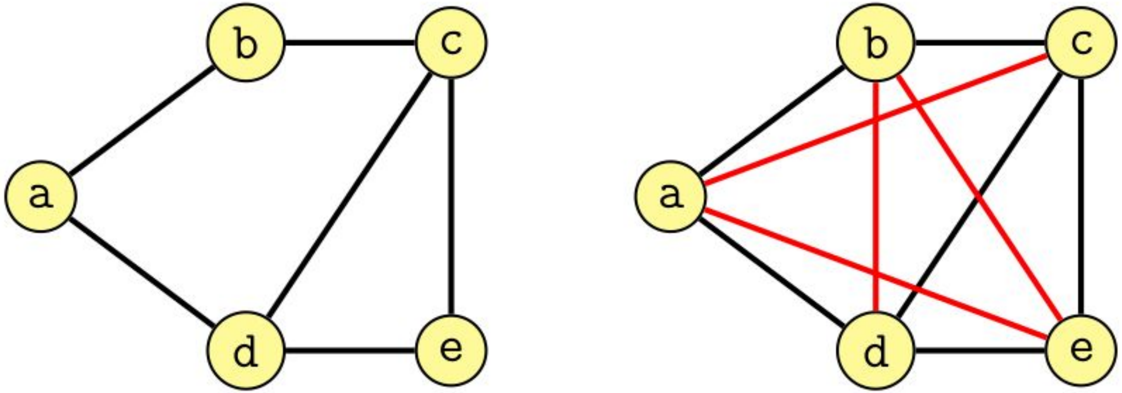




Directed graph
n= 4
m = 12 (=16-4)

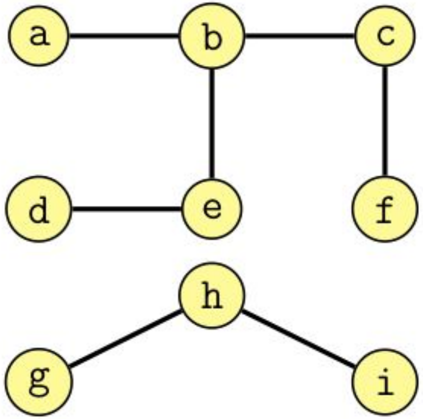
Some special cases

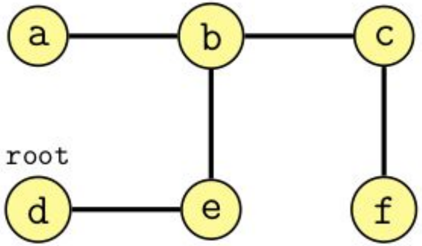
- A graph with an edge between all pairs of nodes is complete
- Informally (there is no agreement on the definitions)
  - A graph with "few" edges is said to be sparse; e.g., graphs with $m = O(n)$, $m = O(n \log n)$
  - A graph with "several" edges is said to be dense; e.g. $m = \Omega(n^2)$



Some special cases

- An unrooted tree is a connected graph with $m = n - 1$
- A rooted tree is a connected graph with $m = n - 1$ in which one node is designated as the root.
- A set of trees is called a forest

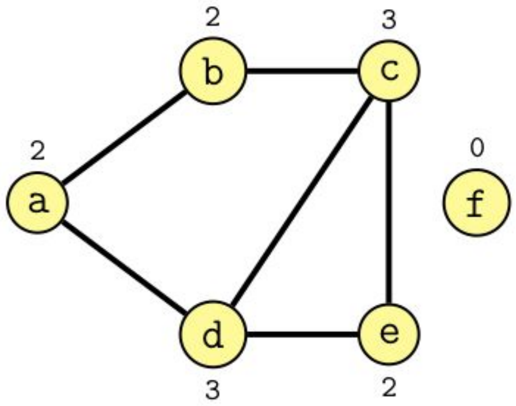




Degree

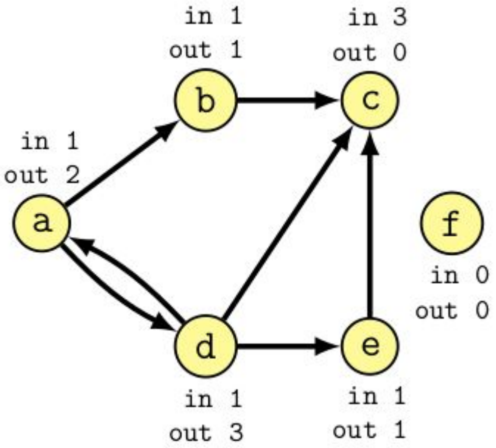
Undirected graphs

The degree of a node is the number of edges incident on it.



Directed graphs

The in-degree (out-degree) of a node is the number of edges incident to (from) it.



Random graphs

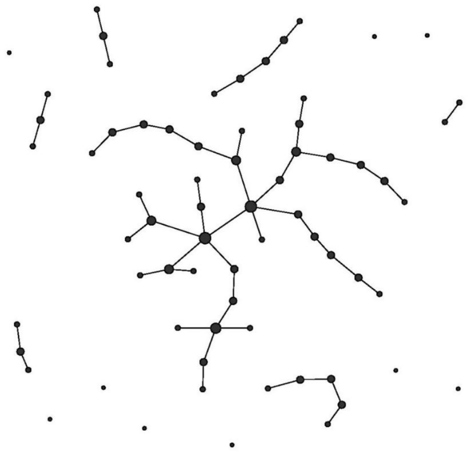
Erdös-Renyi (ER) Model

Create a network with **n nodes** connecting them with **m (undirected) edges** chosen randomly out of the possible **n*(n-1)/2** edges.

The probability of two random nodes to be connected is: **p = 2m / (n *(n − 1))**

The probability of a node to have a **degree k** (approx. Poisson):

$$p(k) \simeq e^{-\langle k \rangle} \frac{\langle k \rangle^k}{k!}$$



E-R graph with p=0.01

Random graphs (1)

Barabasi-Albert (BA) Model

Networks grow: nodes are not fixed but grow as a function of time

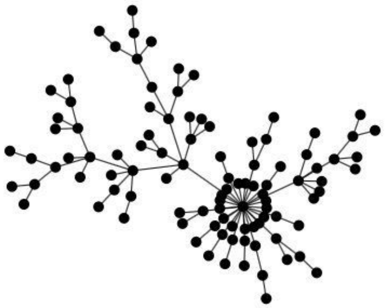
Preferential attachment: the probability that a node gets an edge is proportional to its current degree.

Start from a network with **n nodes** and **m edges** and **add a node at every step**, connecting it to **p<= N** other nodes (with probability depending on their degree).

At time **T** the network will have **n+T nodes** and **m+pT edges**.

The probability of a node to have a **degree k**:

$$p(k) \sim k^{-\gamma_{BA}}$$



Example: scale free networks

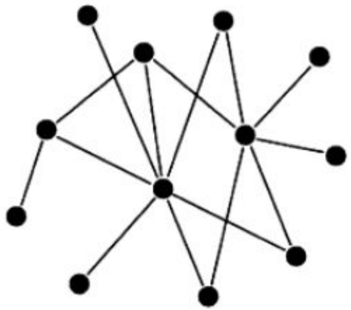
BA networks are **scale free**: <u>many vertices have few links</u> while some (**hubs**) are <u>highly connected</u>

Very robust against failure but vulnerable to intentional attacks

Examples of scale free networks:

- Protein-protein interaction networks
- Signal transduction and transcription networks
- Internet and social relationships

Most highly connected proteins in the cell are the most important for survival

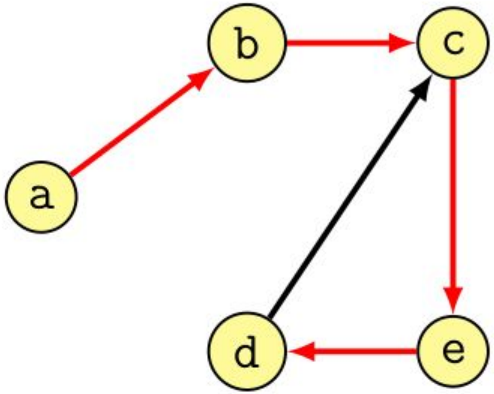


[A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*,286(5439):509–512, 1999]

Definition: Path

Path

In a graph $G = (V, E)$, a path C of length k is a sequence of nodes $u_0, u_1, \ldots, u_k$ such that $(u_i, u_{i+1} \in E)$ for $0 \leq i \leq k-1$.



Example: `a, b, c, e, d`
is a path of length 4

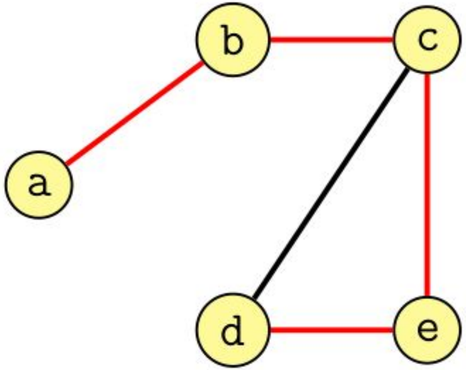
It is also the shortest path between `a` and `d`

Note: a path is said to be simple if all its nodes are distinct

Definition: Path

Path

In a graph $G = (V, E)$, a path C of length k is a sequence of nodes $u_0, u_1, \ldots, u_k$ such that $(u_i, u_{i+1} \in E)$ for $0 \leq i \leq k-1$.



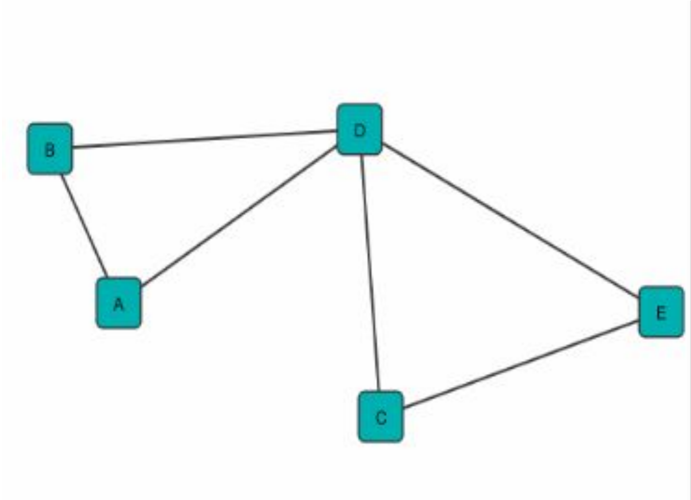
Example: `a, b, c, e, d`
is a path of length 4

Note: a path is said to be simple if all its nodes are distinct

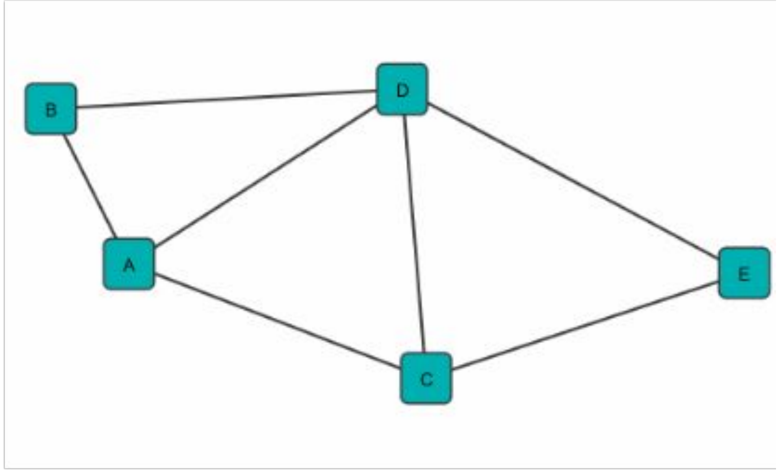
a,b,c,d is the shortest path from a to d

Finding paths...

Eulerian Cycle (undirected graphs)



YES: DABDCED



NO

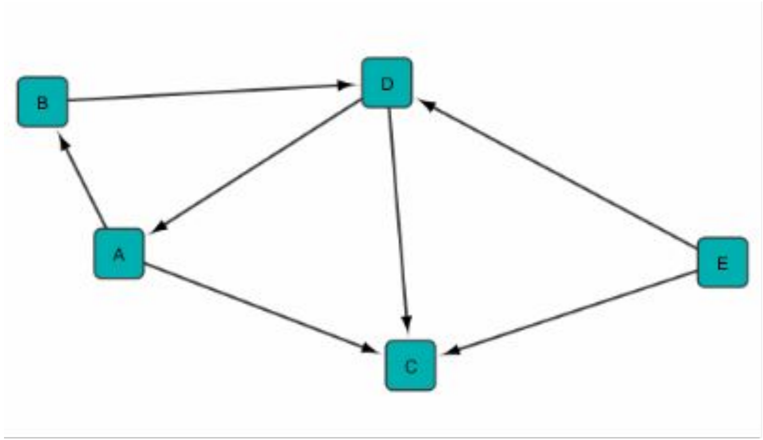
Is it possible to walk around the graph in a way that would involve **crossing each EDGE exactly once getting back to start node**?

If and only if 0 or 2 nodes have an ODD number of edges

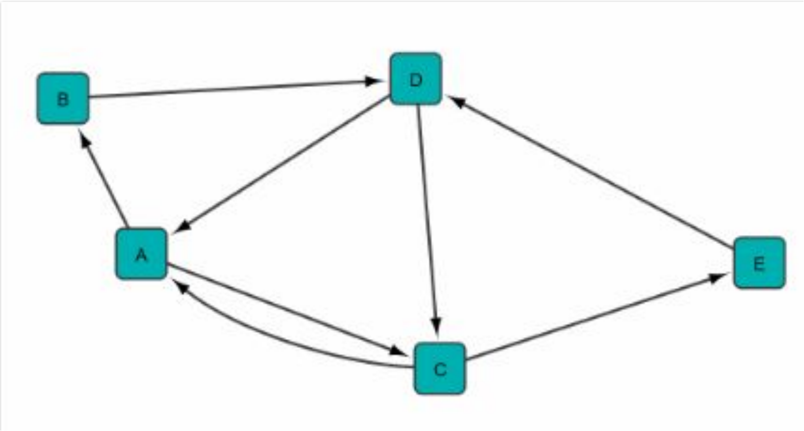
Algorithms exist to find the path in O(n+m)

Finding paths...

Eulerian Cycle (directed graphs)



NO



YES: DCACEDABD

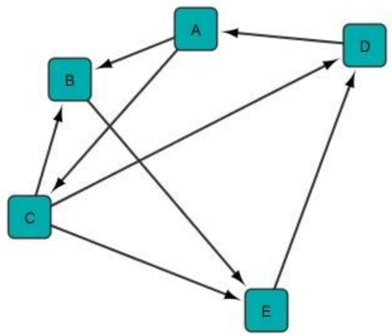
Is it possible to walk around the graph in a way that would involve **crossing each EDGE exactly once getting back to start node**?

If the in-degree and out-degree of all nodes are EQUAL

Algorithms exist to find the path in O(n+m)

Finding paths...

Hamiltonian Cycle (undirected graphs)



YES: ACBEDA

NP-complete problem:

Problems for which there are no polynomial time algorithms known.
IF there was one, then all NP problems would be solved polynomially and P would be equal to NP (P=NP).
Interestingly, it is easy to check if a solution is correct or not (but it is very hard to find such a solution!).

YES, if each node has degree >=n/2 (num nodes, n >3)

Is it possible to walk around the graph in a way that would involve **crossing each NODE exactly once getting back to start node**?

This is a more complex problem. No polynomial solution is currently known!

Graph ADT

In the most general case, graphs are dynamic data structures in which nodes and edges can be added/removed

GRAPH	
Graph( )	% Create a new graph
INT size()	% Returns the number of nodes
SET V()	% Returns the set of all nodes
SET adj(NODE u)	% Returns the set of nodes adjacent to u
insertNode(NODE u)	% Add node u to the graph
insertEdge(NODE u, NODE v)	% Add edge (u, v) to the graph
deleteNode(NODE u)	% Removes node u from the graph
deleteEdge(NODE u, NODE v)	% Removes edge (u, v) from the graph

NOTE: sometimes graphs don't change after being loaded (no delete)

How can we represent a graph?

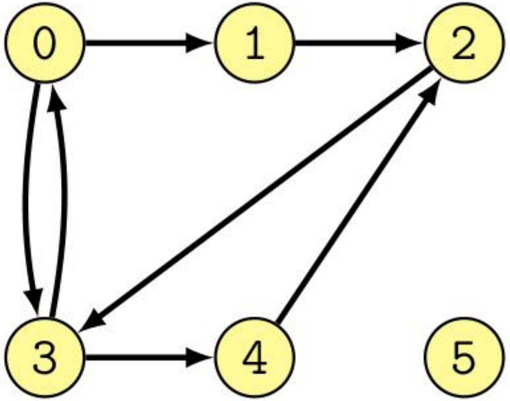
Two possible "classic" implementations

- Adjacency matrix
- Adjacency lists

Adjacency matrix

$$m_{uv} = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases}$$

Space $= n^2$ bits



$$\begin{array}{c c} & \begin{array}{cccccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} \end{array} \\ \begin{array}{c} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{array} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Adjacency matrix

+ : flexible, can put weights on edges

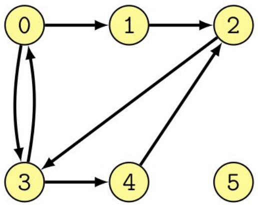
+ : quick to check if edge is present (both ways!)

+ : in undirected graphs, matrix is symmetric
(saves half of the space)

- : in general, it uses a lot of space
(matrix n x n no matter how many edges)

$$m_{uv} = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases}$$

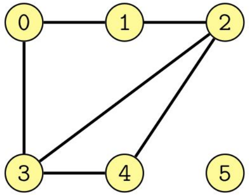
Space $= n^2$ bits



$$\begin{array}{c|c} & \begin{matrix} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} \end{matrix} \\ \begin{matrix} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{matrix} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

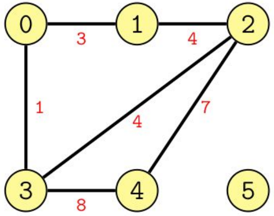
$$m_{uv} = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases}$$

Space $= n^2$ or $n(n-1)/2$



$$\begin{array}{c|c} & \begin{matrix} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} \end{matrix} \\ \begin{matrix} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{matrix} & \begin{pmatrix} & 1 & 0 & 1 & 0 & 0 \\ & & 1 & 0 & 0 & 0 \\ & & & 1 & 1 & 0 \\ & & & & 1 & 0 \\ & & & & & 0 \\ & & & & & \end{pmatrix} \end{array}$$

- Edges may be associated with a weight (cost, profit, etc.)
- The weight is associated through a cost function $w : V \times V \rightarrow \mathbb{R}$
- If there is no edge between two vertices u, v, $w(u,v) = +\infty$

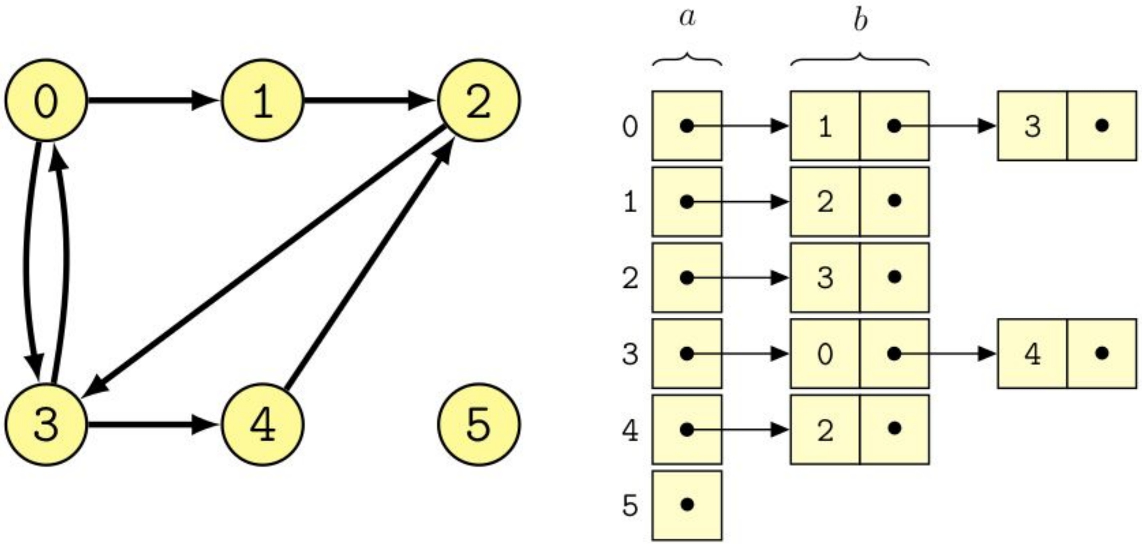


$$\begin{array}{c|c} & \begin{matrix} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} \end{matrix} \\ \begin{matrix} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{matrix} & \begin{pmatrix} & 3 & 0 & 1 & 0 & 0 \\ & & 4 & 0 & 0 & 0 \\ & & & 4 & 7 & 0 \\ & & & & 8 & 0 \\ & & & & & 0 \\ & & & & & \end{pmatrix} \end{array}$$

Adjacency list

$G.\mathsf{adj}(u) = \{v | (u, v) \in E\}$

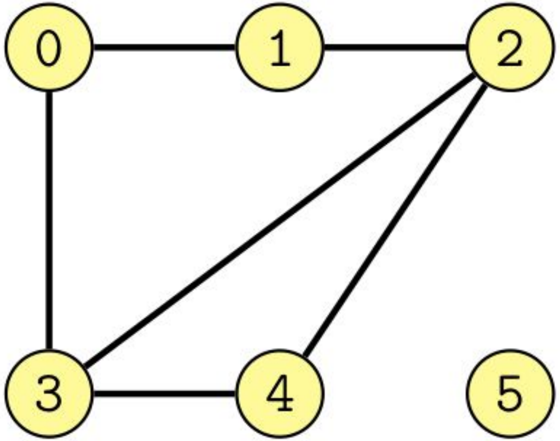
Space $= an + bm$ bits

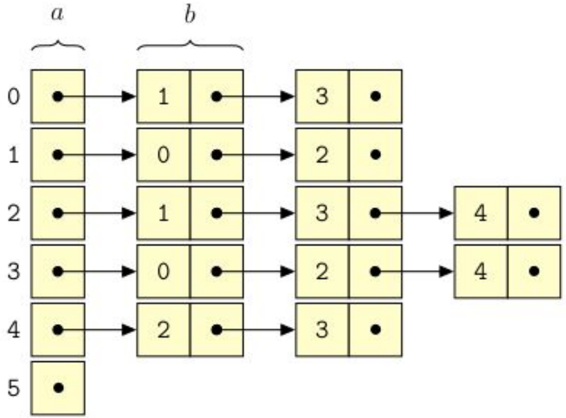


Adjacency list: undirected graph

$$G.\mathsf{adj}(u) = \{v \mid (u, v) \in E\}$$

$$\text{Space} = an + 2 \cdot bm$$





Adjacency list

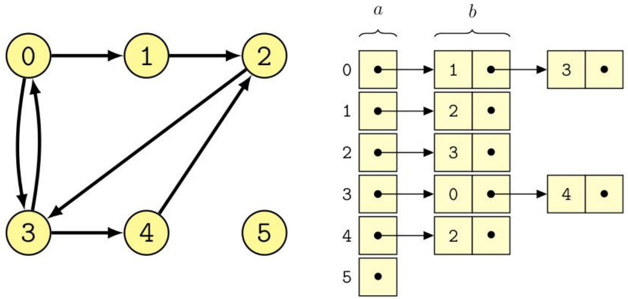
+: flexible, nodes can be complex objects
(ex. node1.list_add(node2);)

+: uses less space

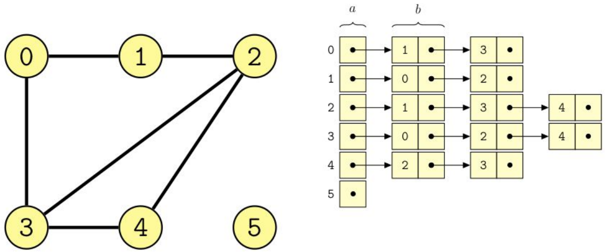
: checking presence of an edge is in general slower
(requires going through the list of source node)

-: getting all incoming edges of a node is slow
(requires going through all nodes!)
Workaround: store another list with all “IN”-linking nodes

$G.\mathsf{adj}(u) = \{v | (u, v) \in E\}$ Space $= an + bm$ bits



$G.\mathsf{adj}(u) = \{v | (u, v) \in E\}$ Space $= an + 2 \cdot bm$



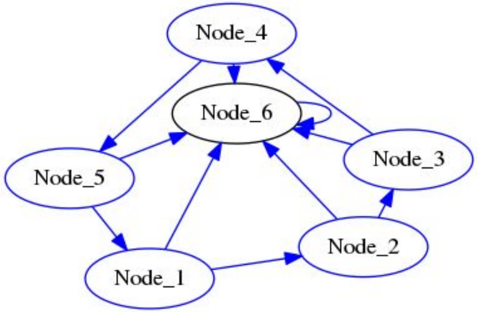
Possible implementations

Structure	Java	C++	Python
Linked list	LinkedList	list	
Static vector	[]	[]	[]
Dynamic vector	ArrayList	vector	list
Set	HashSet TreeSet	set	set
Dictionary	HashMap TreeMap	map	dict

Both the concepts of adjacency matrix and adjacency list can be implemented in several ways.
Our simple implementation will use a **dictionary**

Graph as adjacency matrix: exercise

```python
class DiGraphAsAdjacencyMatrix:
    def __init__(self):
        #would be better a set, but I need an index
        self.__nodes = list()
        self.__matrix = list()

    def __len__(self):
        """gets the number of nodes"""
        return len(self.__nodes)

    def nodes(self):
        return self.__nodes

    def matrix(self):
        return self.__matrix

    def __str__(self):
        #TODO
        pass

    def insertNode(self, node):
        #TODO
        pass

    def insertEdge(self, node1, node2, weight):
        #TODO
        pass

    def deleteEdge(self, node1,node2):
        """removing an edge means to set its
        corresponding place in the matrix to 0"""
        #TODO
        pass

    def deleteNode(self, node):
        """removing a node means removing
        its corresponding row and column in the matrix"""
        #TODO
        pass

    def adjacent(self, node, incoming = True):
        #TODO
        pass

    def edges(self):
        #TODO
        pass
```

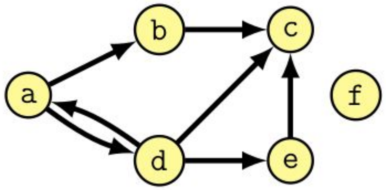


Nodes:
['Node_1', 'Node_2', 'Node_3', 'Node_4', 'Node_5', 'Node_6']
Matrix:
[[0, 0.5, 0, 0, 0, 1], [0, 0, 0.5, 0, 0, 1], [0, 0, 0, 0.5, 0, 1], [0, 0, 0, 0, 0.5, 1], [0.5, 0, 0, 0, 0, 1], [0, 0, 0, 0, 0, 1]]

Output of print(G):

	Node_1	Node_2	Node_3	Node_4	Node_5	Node_6
Node_1	0	0.5	0	0	0	1
Node_2	0	0	0.5	0	0	1
Node_3	0	0	0	0.5	0	1
Node_4	0	0	0	0	0.5	1
Node_5	0.5	0	0	0	0	1
Node_6	0	0	0	0	0	1

Weighted Graph (adj list as a dict of dicts)




```python
class Graph:

    # initializer, nodes are private!
    def __init__(self):
        self.__nodes = dict()

    #returns the size of the Graph
    #accessible through len(Graph)
    def __len__(self):
        return len(self.__nodes)

    #returns the nodes
    def V(self):
        return self.__nodes.keys()

    #a generator of nodes to access all of them
    #once (not a very useful example!)
    def node_iterator(self):
        for n in self.__nodes.keys():
            yield n

    #a generator of edges (as triplets (u,v,w)) to access all of them
    def edge_iterator(self):
        for u in self.__nodes:
            for v in self.__nodes[u]:
                yield (u,v,self.__nodes[u][v])

    #returns all the adjacent nodes of node
    #as a dictionary with key as the other node
    #and value the weight
    def adj(self,node):
        if node in self.__nodes.keys():
            return self.__nodes[node]

    #adds the node to the graph
    def insert_node(self, node):
        if node not in self.__nodes:
            self.__nodes[node] = dict()
```

```python
    #adds the edge startN --> endN with weight w
    #that has 0 as default
    def insert_edge(self, startN, endN, w = 0):
        #does nothing if already in
        self.insert_node(startN)
        self.insert_node(endN)
        self.__nodes[startN][endN] = w

    #converts the graph into a string
    def __str__(self):
        out_str = "Nodes:\n" + ",".join(self.__nodes)
        out_str +="\nEdges:\n"
        for u in self.__nodes:
            for v in self.__nodes[u]:
                out_str +="{} --{}--> {}\n".format(u,self.__nodes[u][v],v )
            if len(self.__nodes[u]) == 0:
                out_str +="{}\n".format(u)
        return out_str
```

```python
if __name__ == "__main__":
    G = Graph()
    for u,v in [ ('a', 'b'), ('a', 'd'), ('b', 'c'),
                 ('d', 'a'), ('d', 'c'), ('d', 'e'), ('e', 'c') ]:
        G.insert_edge(u,v)
    for edge in G.edge_iterator():
        print("{} --{}--> {}".format(edge[0],
                                     edge[1],
                                     edge[2]))
    G.insert_node('f')
    print("\nG has {} nodes:".format(len(G)))
    for node in G.node_iterator():
        print("{}".format(node), end= " ")
    print()
    print(G)
```

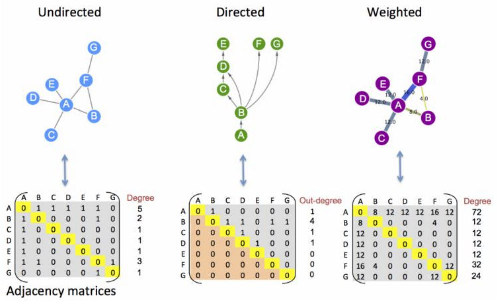
```
a --b--> 0
a --d--> 0
b --c--> 0
d --a--> 0
d --c--> 0
d --e--> 0
e --c--> 0

G has 6 nodes:
a b d c e f
Nodes:
a,b,d,c,e,f
Edges:
a --0--> b
a --0--> d
b --0--> c
d --0--> a
d --0--> c
d --0--> e
c
e --0--> c
f
```

for simplicity nodes are strings (can make them objects as an exercise)

Summary

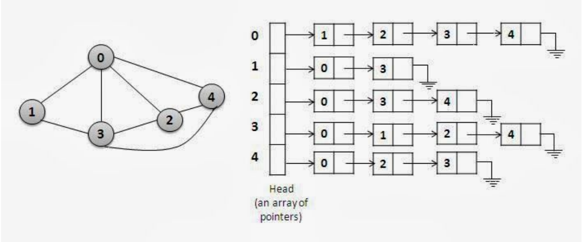
Adjacency matrix

- Required space $O(n^2)$
- To check whether u is adjacent to v requires $O(1)$ time
- Ideal for dense graphs



Adjacency lists/vectors

- Required space $O(n + m)$
- To check whether u is adjacent to v requires $O(n)$
- Ideal for sparse graphs



Adjacency List Representation of Graph

Iterating through nodes/edges

Equivalent ways of looping through nodes and edges

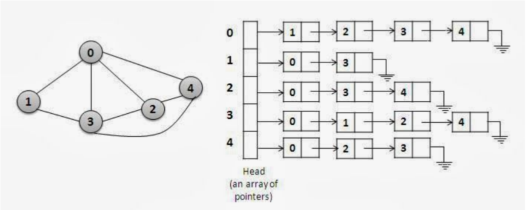
```python
for node in G.V():
    #do something with the node


for u in G.V():
    #for all starting nodes u
    for v in G.adj(u):
        #for all ending nodes v
        #do something with (u,v)
```

```python
for node in G.node_iterator():
    #do something with the node

for edge in G.edge_iterator():
    #do something with the edge
```

How much do these operations cost? (n nodes, m edges)

$$
\begin{array}{c c}
 & \begin{array}{cccccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} \end{array} \\
\begin{array}{c} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{array} &
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$

- Looping through nodes is O(n)
- Looping through edges is:
    - O(m + n) with adjacency lists and variants
    - O(n^2) with adjacency matrices



Adjacency List Representation of Graph

Graph traversal

Problem definition

Given a graph $G = (V, E)$ and a vertex $r \in V$ (root), visit exactly once all the vertexes of the graph that can be reached from r

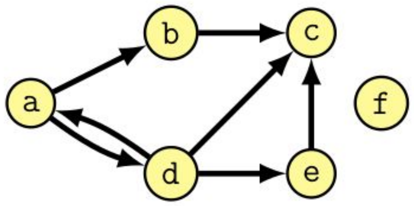
Naive idea, just iterate through the nodes and edges with:

```
for u in G.V():
    #for all starting nodes u
    for v in G.adj(u):
        #for all ending nodes v
        #do something with (u,v)
```

or

```
for edge in G.edge_iterator():
    #do something with the edge
```

but this does not take into account the topology of the graph and is still O(n + m)



OK in some cases, but not what we are looking for!

Graph traversal

Problem definition

Given a graph $G = (V, E)$ and a vertex $r \in V$ (root), visit exactly once all the vertexes of the graph that can be reached from r

As in the case of trees, two possible methods:

- Breadth first search (BFS)
- Depth first search (DFS)

Graph traversal

Problem definition

Given a graph $G = (V, E)$ and a vertex $r \in V$ (root), visit exactly once all the vertexes of the graph that can be reached from r

As in the case of trees, two possible methods:

- Breadth first search (BFS)
- Depth first search (DFS)

but graphs are more complicated that trees (these are Direct Acyclic Graphs)

no matter what,
beware of cycles!
Hint: mark visited nodes

Graph traversal: BFS

Problem definition

Given a graph $G = (V, E)$ and a vertex $r \in V$ (root), visit exactly once all the vertexes of the graph that can be reached from r

Breadth-first search (BFS)

Traverse the graph by visiting the nodes by levels: first by visiting the nodes at distance 1 from the source, then distance 2, etc.

- Application: compute the shortest paths from a single source

BFS, goals

To visit nodes at increasing distances from the source

- Visit nodes at distance k before visiting nodes at distance $k+1$

Generate a breadth-first tree

- To generate a tree containing all the nodes reachable from r and such that the path between the root r and the node in the tree corresponds to a shortest path in the graph

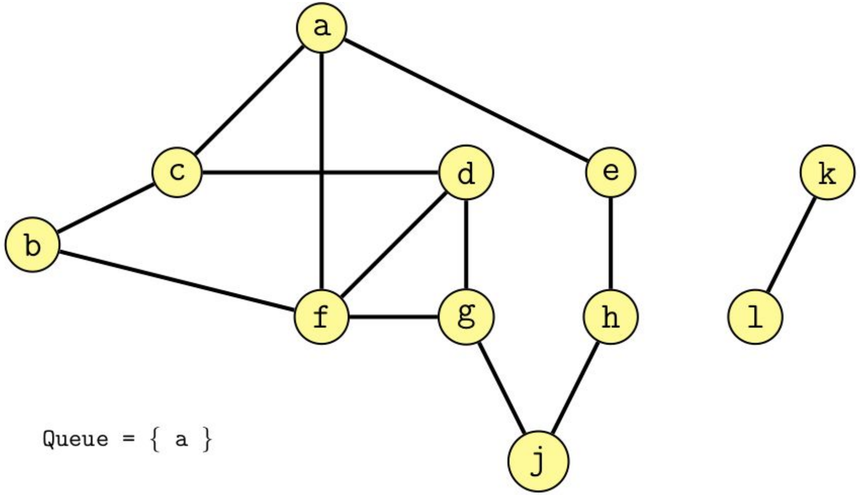
Compute the shortest path from s to all the other reachable nodes

- Distance measured as the number of edges to be traversed

Graph traversal

Warning. Wrong code!!!

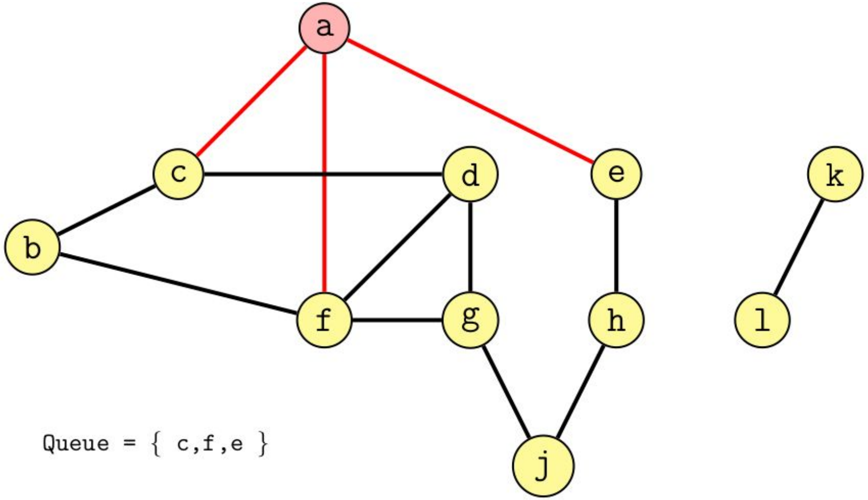
```python
from collections import deque()

def BFS(node):
    Q = deque()
    if node != None:
        Q.append(node)

    while len(Q) > 0:
        curNode = Q.popleft()
        if curNode != None:
            print("{}".format(curNode))
            for v in G.adj(curNode):
                Q.append(v)
```



Graph traversal

Warning. Wrong code!!!

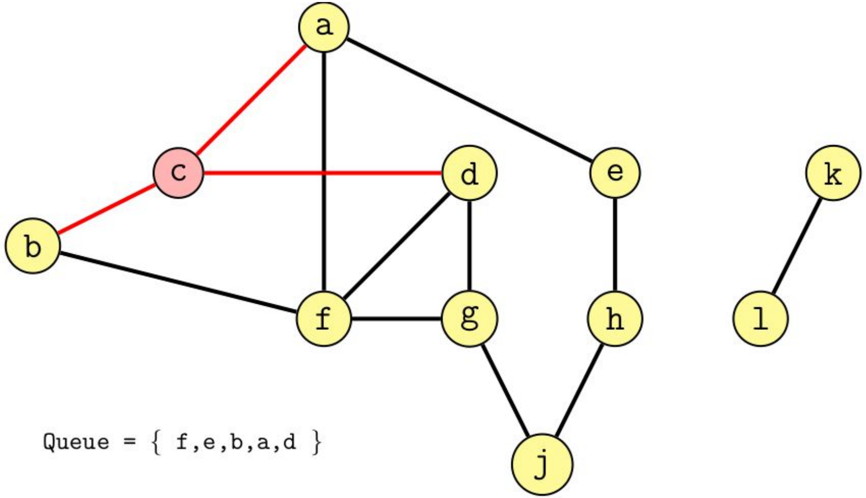
```
from collections import deque()

def BFS(node):
    Q = deque()
    if node != None:
        Q.append(node)

    while len(Q) > 0:
        curNode = Q.popleft()
        if curNode != None:
            print("{}".format(curNode))
            for v in G.adj(curNode):
                Q.append(v)
```



Graph traversal

Warning. Wrong code!!!

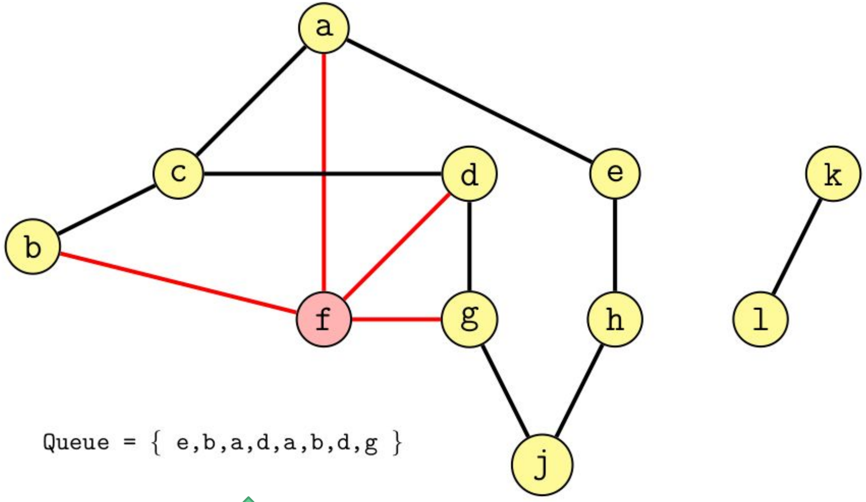
```
from collections import deque()

def BFS(node):
    Q = deque()
    if node != None:
        Q.append(node)

    while len(Q) > 0:
        curNode = Q.popleft()
        if curNode != None:
            print("{}".format(curNode))
            for v in G.adj(curNode):
                Q.append(v)
```



Graph traversal

Warning. Wrong code!!!

```
from collections import deque()

def BFS(node):
    Q = deque()
    if node != None:
        Q.append(node)

    while len(Q) > 0:
        curNode = Q.popleft()
        if curNode != None:
            print("{}".format(curNode))
            for v in G.adj(curNode):
                Q.append(v)
```



Queue = { e,b,a,d,a,b,d,g }

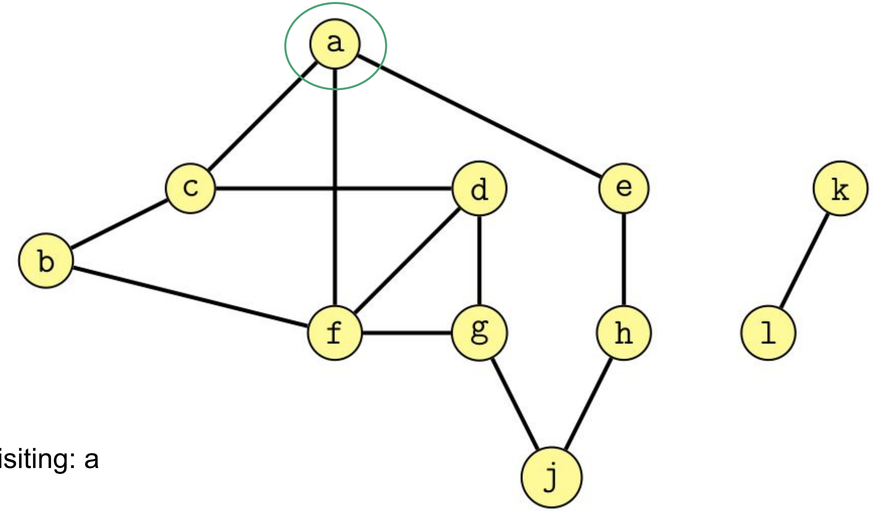
even though we can avoid adding elements already in the Queue, this never gets empty!
→ **infinite loop!**

Graph traversal: BFS

```
from collections import deque

class Graph:

"""..."""

    def BFS(self, node):
        Q = deque()
        Q.append(node)
        visited = set()
        visited.add(node)
        print("visiting: {}".format(node))

        while len(Q) > 0:
            curNode = Q.popleft()
            #do something with curNode
            for n in self.adj(curNode):
                #do something with edge (curNode, n)
                if n not in visited:
                    Q.append(n)
                    visited.add(n)
                    print("visiting: {}".format(n))

            print("visited: {}".format(visited))
            print("Q: {}".format(list(Q)))
```

enqueue

dequeue



visiting: a

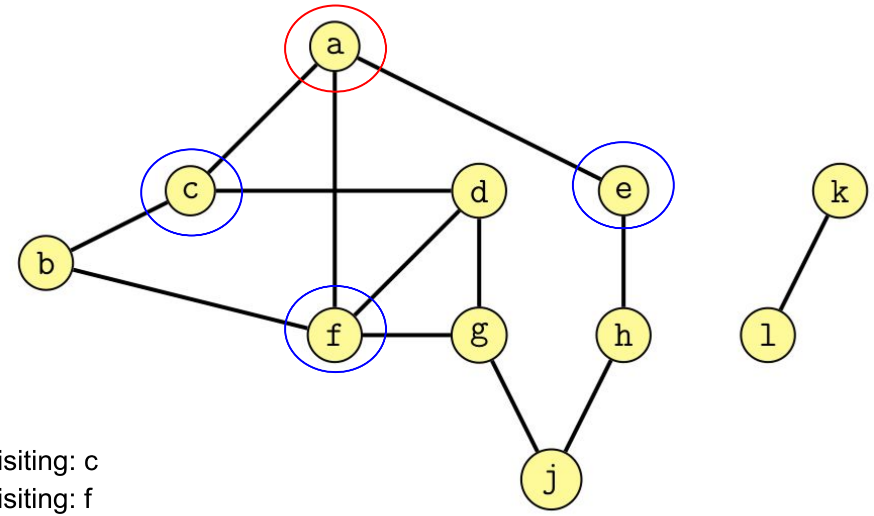
visited: {'a'}

Q: ['a']

DFS visit: a

Graph traversal: BFS

```python
from collections import deque

class Graph:

"""..."""

    def BFS(self, node):
        Q = deque()
        Q.append(node)
        visited = set()
        visited.add(node)
        print("visiting: {}".format(node))

        while len(Q) > 0:
            curNode = Q.popleft()
            #do something with curNode
            for n in self.adj(curNode):
                #do something with edge (curNode, n)
                if n not in visited:
                    Q.append(n)
                    visited.add(n)
                    print("visiting: {}".format(n))

            print("visited: {}".format(visited))
            print("Q: {}".format(list(Q)))
```



visiting: c
visiting: f
visiting: e

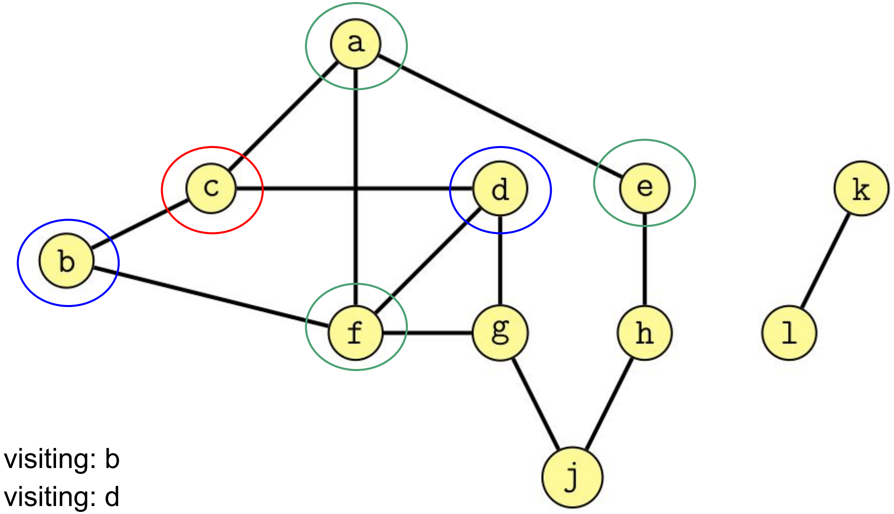
visited: {'e', 'f', 'c', 'a'}

Q: ['c', 'f', 'e'] → a

DFS visit: a, c, f, e

Graph traversal: BFS

```python
from collections import deque

class Graph:

"""..."""

    def BFS(self, node):
        Q = deque()
        Q.append(node)
        visited = set()
        visited.add(node)
        print("visiting: {}".format(node))

        while len(Q) > 0:
            curNode = Q.popleft()
            #do something with curNode
            for n in self.adj(curNode):
                #do something with edge (curNode, n)
                if n not in visited:
                    Q.append(n)
                    visited.add(n)
                    print("visiting: {}".format(n))

            print("visited: {}".format(visited))
            print("Q: {}".format(list(Q)))
```

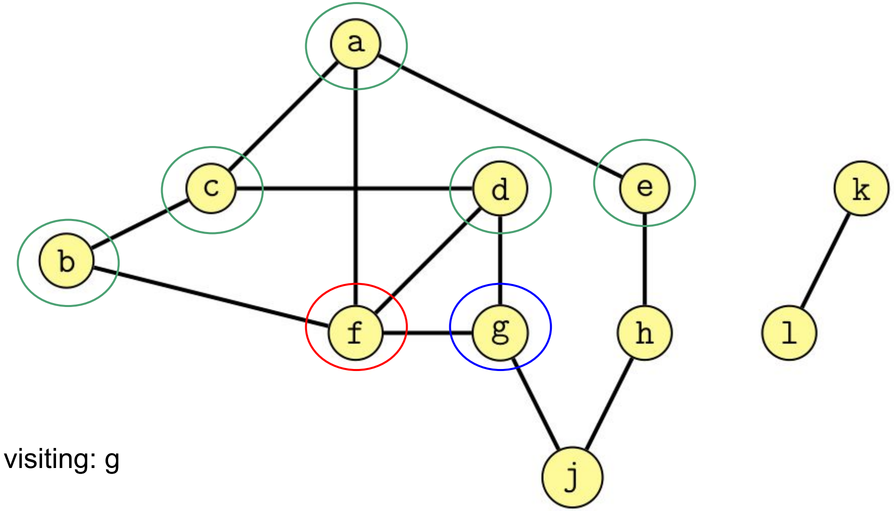


visiting: b
visiting: d

visited: {'d', 'b', 'a', 'c', 'e', 'f'}
Q: ['f', 'e', 'b', 'd'] → c
DFS visit: a, c, f, e, b, d

Graph traversal: BFS

```
from collections import deque

class Graph:

"""..."""

    def BFS(self, node):
        Q = deque()
        Q.append(node)
        visited = set()
        visited.add(node)
        print("visiting: {}".format(node))

        while len(Q) > 0:
            curNode = Q.popleft()
            #do something with curNode
            for n in self.adj(curNode):
                #do something with edge (curNode, n)
                if n not in visited:
                    Q.append(n)
                    visited.add(n)
                    print("visiting: {}".format(n))

            print("visited: {}".format(visited))
            print("Q: {}".format(list(Q)))
```



visiting: g

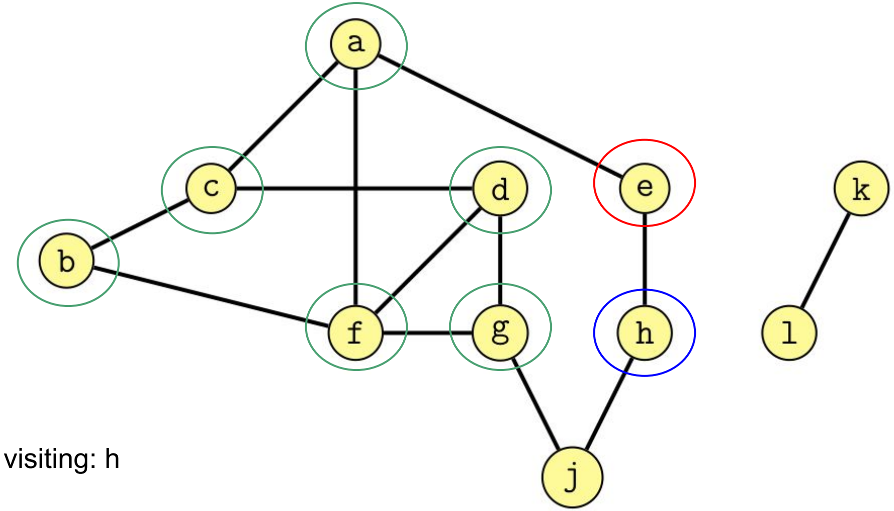
visited: {'d', 'b', 'a', 'g', 'c', 'e', 'f'}

Q: ['e', 'b', 'd', 'g'] → f

DFS visit: a, c, f, e, b, d, g

Graph traversal: BFS

```
from collections import deque

class Graph:

"""..."""

    def BFS(self, node):
        Q = deque()
        Q.append(node)
        visited = set()
        visited.add(node)
        print("visiting: {}".format(node))

        while len(Q) > 0:
            curNode = Q.popleft()
            #do something with curNode
            for n in self.adj(curNode):
                #do something with edge (curNode, n)
                if n not in visited:
                    Q.append(n)
                    visited.add(n)
                    print("visiting: {}".format(n))

            print("visited: {}".format(visited))
            print("Q: {}".format(list(Q)))
```



visiting: h

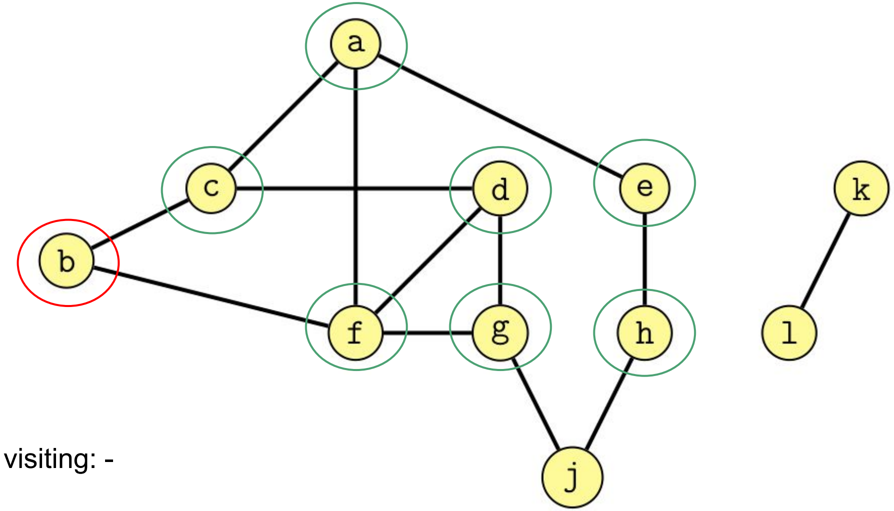
visited: {'d', 'b', 'h', 'a', 'g', 'c', 'e', 'f'}

Q: ['b', 'd', 'g', 'h'] → e

DFS visit: a, c, f, e, b, d, g, h

Graph traversal: BFS

```python
from collections import deque

class Graph:

"""..."""

    def BFS(self, node):
        Q = deque()
        Q.append(node)
        visited = set()
        visited.add(node)
        print("visiting: {}".format(node))

        while len(Q) > 0:
            curNode = Q.popleft()
            #do something with curNode
            for n in self.adj(curNode):
                #do something with edge (curNode, n)
                if n not in visited:
                    Q.append(n)
                    visited.add(n)
                    print("visiting: {}".format(n))

            print("visited: {}".format(visited))
            print("Q: {}".format(list(Q)))
```



visiting: -

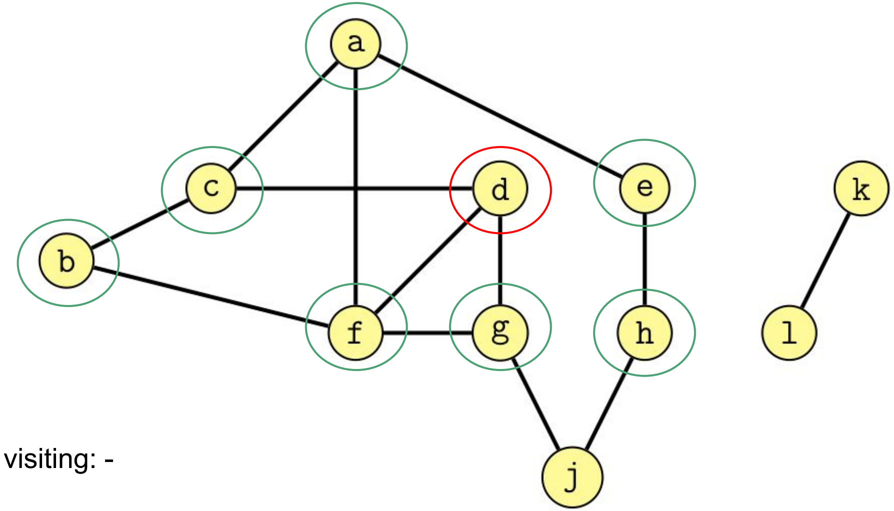
visited: {'d', 'b', 'h', 'a', 'g', 'c', 'e', 'f'}

Q: ['d', 'g', 'h'] → b

DFS visit: a, c, f, e, b, d, g, h

Graph traversal: BFS

```python
from collections import deque

class Graph:

"""..."""

    def BFS(self, node):
        Q = deque()
        Q.append(node)
        visited = set()
        visited.add(node)
        print("visiting: {}".format(node))

        while len(Q) > 0:
            curNode = Q.popleft()
            #do something with curNode
            for n in self.adj(curNode):
                #do something with edge (curNode, n)
                if n not in visited:
                    Q.append(n)
                    visited.add(n)
                    print("visiting: {}".format(n))

            print("visited: {}".format(visited))
            print("Q: {}".format(list(Q)))
```



visiting: -

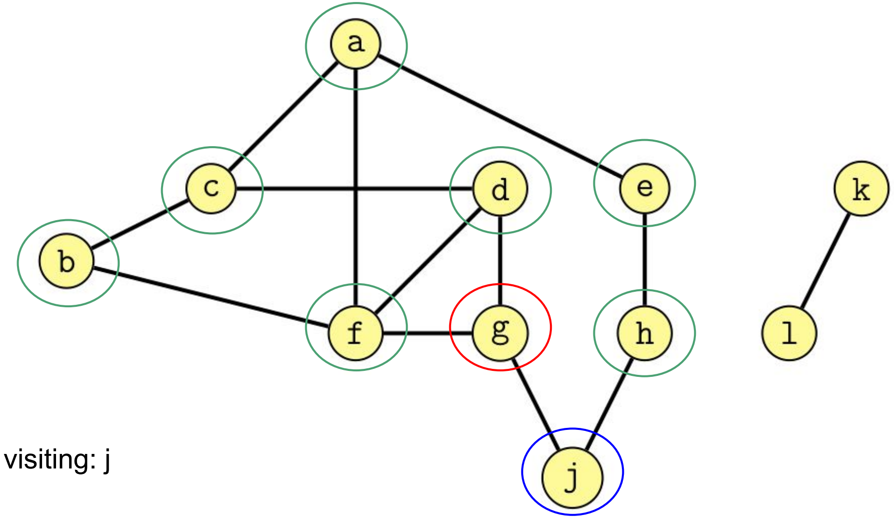
visited: {'d', 'b', 'h', 'a', 'g', 'c', 'e', 'f'}

Q: ['g', 'h'] → d

DFS visit: a, c, f, e, b, d, g, h

Graph traversal: BFS

```python
from collections import deque

class Graph:

"""..."""

    def BFS(self, node):
        Q = deque()
        Q.append(node)
        visited = set()
        visited.add(node)
        print("visiting: {}".format(node))

        while len(Q) > 0:
            curNode = Q.popleft()
            #do something with curNode
            for n in self.adj(curNode):
                #do something with edge (curNode, n)
                if n not in visited:
                    Q.append(n)
                    visited.add(n)
                    print("visiting: {}".format(n))

            print("visited: {}".format(visited))
            print("Q: {}".format(list(Q)))
```



visiting: j

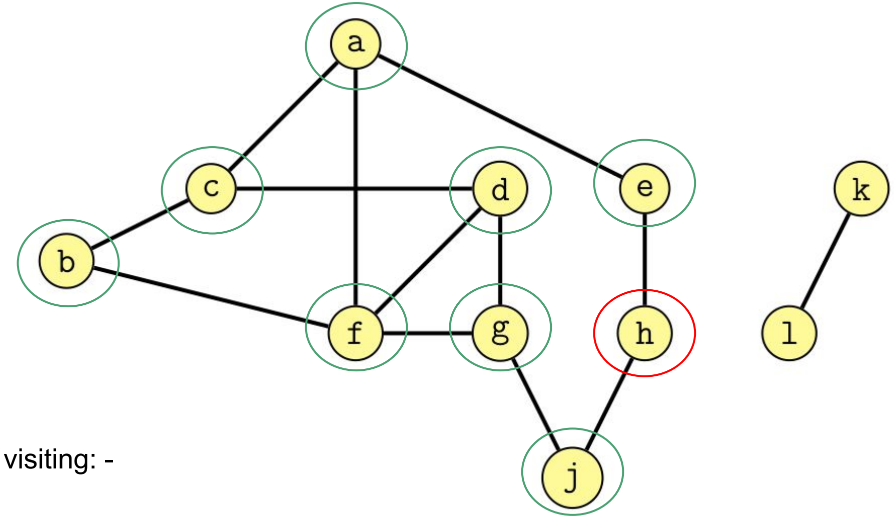
visited: {'d', 'b', 'j', 'h', 'a', 'g', 'c', 'e', 'f'}

Q: ['h', 'j'] → g

DFS visit: a, c, f, e, b, d, g, h, j

Graph traversal: BFS

```python
from collections import deque

class Graph:

"""..."""

    def BFS(self, node):
        Q = deque()
        Q.append(node)
        visited = set()
        visited.add(node)
        print("visiting: {}".format(node))

        while len(Q) > 0:
            curNode = Q.popleft()
            #do something with curNode
            for n in self.adj(curNode):
                #do something with edge (curNode, n)
                if n not in visited:
                    Q.append(n)
                    visited.add(n)
                    print("visiting: {}".format(n))

            print("visited: {}".format(visited))
            print("Q: {}".format(list(Q)))
```



visiting: -

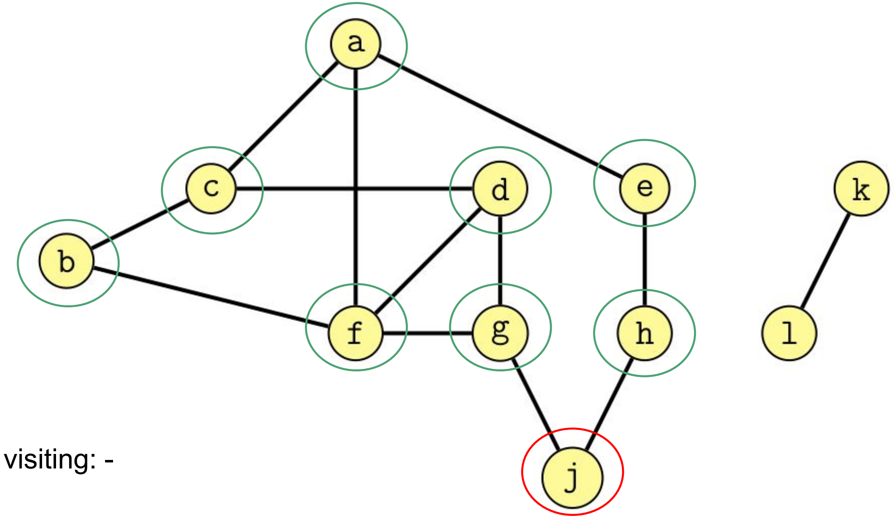
visited: {'d', 'b', 'j', 'h', 'a', 'g', 'c', 'e', 'f'}

Q: ['j'] → h

DFS visit: a, c, f, e, b, d, g, h, j

Graph traversal: BFS

```python
from collections import deque

class Graph:

"""..."""

    def BFS(self, node):
        Q = deque()
        Q.append(node)
        visited = set()
        visited.add(node)
        print("visiting: {}".format(node))

        while len(Q) > 0:
            curNode = Q.popleft()
            #do something with curNode
            for n in self.adj(curNode):
                #do something with edge (curNode, n)
                if n not in visited:
                    Q.append(n)
                    visited.add(n)
                    print("visiting: {}".format(n))

            print("visited: {}".format(visited))
            print("Q: {}".format(list(Q)))
```



visiting: -

Node	Dist from a
a	0
c	1
f	1
e	1
b	2
d	2
g	2
h	2
j	3

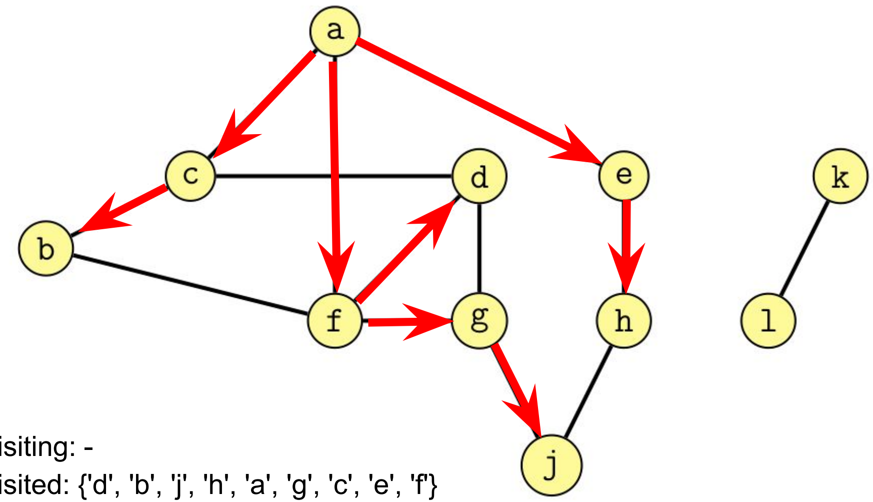
visited: {'d', 'b', 'j', 'h', 'a', 'g', 'c', 'e', 'f'}

Q: [] → **DONE** → j

DFS visit: a, c, f, e, b, d, g, h, j

Graph traversal: BFS tree of the graph

```
from collections import deque

class Graph:

"""..."""

    def BFS(self, node):
        Q = deque()
        Q.append(node)
        visited = set()
        visited.add(node)
        print("visiting: {}".format(node))

        while len(Q) > 0:
            curNode = Q.popleft()
            #do something with curNode
            for n in self.adj(curNode):
                #do something with edge (curNode, n)
                if n not in visited:
                    Q.append(n)
                    visited.add(n)
                    print("visiting: {}".format(n))

            print("visited: {}".format(visited))
            print("Q: {}".format(list(Q)))
```



visiting: -
visited: {'d', 'b', 'j', 'h', 'a', 'g', 'c', 'e', 'f'}
Q: []
DONE!

BFS from a: c, f, e, b, d, g, h, j

This can be done by storing a pointer to parents!

Graph traversal: BFS complexity

Complexity: $O(n + m)$

- every node is inserted in the queue at most once;
- whenever a node is extracted all its edges are analyzed once and only once;
- number of edges analyzed:

$$m = \sum_{u \in V} out\_degree(u)$$

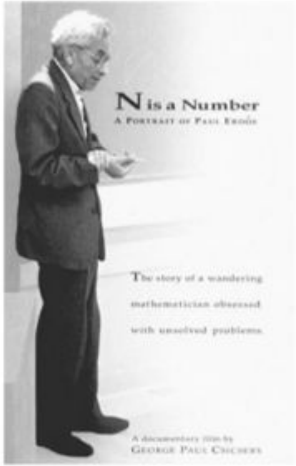
BFS: application. Shortest path

Paul Erdös (1913-1996)

- Mathematician
- 1500+ papers, 500+ co-authors

Erdös number

- Erdös has $erdos = 0$
- The co-authors of Erdös have $erdos = 1$
- If X is co-author of someone with $erdos = k$, but is not co-author of someone with $erdos < k$, then X has $erdos = k + 1$
- People who are not reached by this definition have $erdos = +\infty$



Find the path between two authors:

Luca Bianco | Paul Erdős

Luca Bianco
co-authored 11 papers with
Vincenzo Manca
co-authored 1 paper with
Henning Fernau
co-authored 1 paper with
Zsolt Tuza
co-authored 7 papers with
Paul Erdős
distance = 4

for fun: https://www.csauthors.net/distance

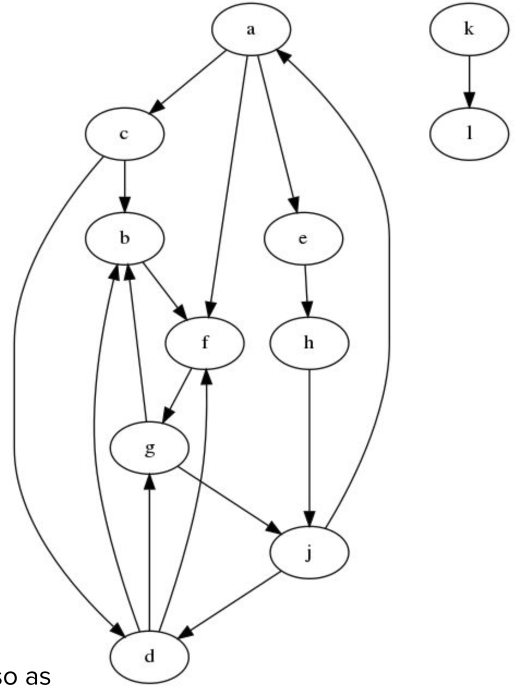
BFS: application. Shortest distance/Shortest path

```python
from collections import deque
import math

class Graph:

"""..."""

    #computes the distance from root of all nodes
    def get_distance(self, root):
        distances = dict()
        parents = dict()
        for node in self.node_iterator():
            distances[node] = math.inf
            parents[node] = -1
        Q = deque()
        Q.append(root)
        distances[root] = 0
        parents[root] = root
        while len(Q) > 0:
            curNode = Q.popleft()
            for n in self.adj(curNode):
                if distances[n] == math.inf:
                    distances[n] = distances[curNode] + 1
                    parents[n] = curNode
                    Q.append(n)
        return (distances,parents)
```

Initially
all distances: +∞
all parents: -1

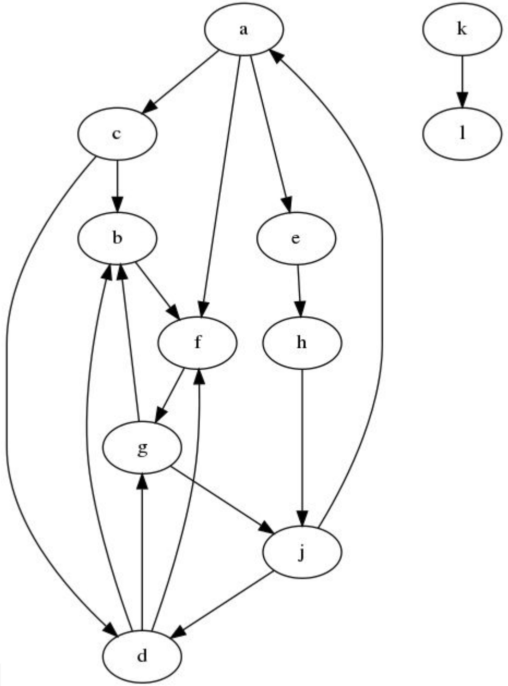
distance root <-> root = 0
parent of root = root

distances is used also as 'visited'
if not set, distance node: distance of parent +1



BFS: application. Shortest distance/Shortest path

```python
from collections import deque
import math

class Graph:

"""..."""

    #computes the distance from root of all nodes
    def get_distance(self, root):
        distances = dict()
        parents = dict()
        for node in self.node_iterator():
            distances[node] = math.inf
            parents[node] = -1
        Q = deque()
        Q.append(root)
        distances[root] = 0
        parents[root] = root
        while len(Q) > 0:
            curNode = Q.popleft()
            for n in self.adj(curNode):
                if distances[n] == math.inf:
                    distances[n] = distances[curNode] + 1
                    parents[n] = curNode
                    Q.append(n)
        return (distances,parents)
```

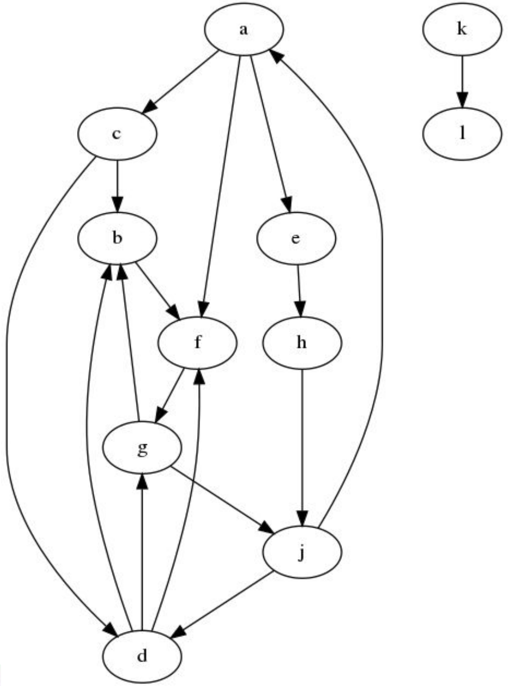



```python
D, P = G1.get_distance('a')
print("Distances from 'a': {}".format(D))
print("All parents: {}".format(P))
```

Distances from 'a': {'a': 0, 'c': 1, 'f': 1, 'e': 1, 'b': 2, 'd': 2, 'g': 2, 'j': 3, 'h': 2, 'k': inf, 'l': inf}
All parents: {'a': 'a', 'c': 'a', 'f': 'a', 'e': 'a', 'b': 'c', 'd': 'c', 'g': 'f', 'j': 'g', 'h': 'e', 'k': -1, 'l': -1}

BFS: application. Shortest distance/Shortest path

```python
from collections import deque
import math

class Graph:

"""..."""

    #computes the distance from root of all nodes
    def get_distance(self, root):
        distances = dict()
        parents = dict()
        for node in self.node_iterator():
            distances[node] = math.inf
            parents[node] = -1
        Q = deque()
        Q.append(root)
        distances[root] = 0
        parents[root] = root
        while len(Q) > 0:
            curNode = Q.popleft()
            for n in self.adj(curNode):
                if distances[n] == math.inf:
                    distances[n] = distances[curNode] + 1
                    parents[n] = curNode
                    Q.append(n)
        return (distances,parents)
```



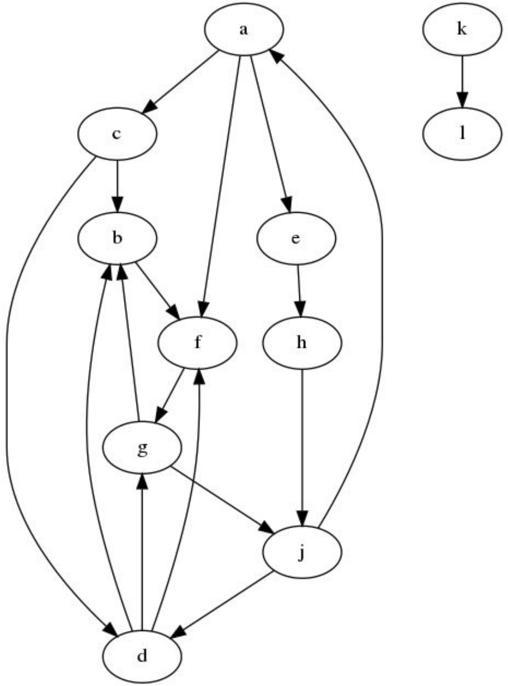

```python
D, P = G2.get_distance('b')
print("Distances from 'b': {}".format(D))
print("All parents: {}".format(P))
```

Distances from 'b': {'a': 4, 'c': 5, 'f': 1, 'e': 5, 'b': 0, 'd': 4, 'g': 2, 'j': 3, 'h': 6, 'k': inf, 'l': inf}
All parents: {'a': 'j', 'c': 'a', 'f': 'b', 'e': 'a', 'b': 'b', 'd': 'j', 'g': 'f', 'j': 'g', 'h': 'e', 'k': -1, 'l': -1}

BFS: application. Shortest distance/Shortest path

printing the shortest path...

```python
def printPath(startN, endN, parents):
    outPath = str(endN)
    #this assumes all the nodes are in the
    #parents structure
    curN = endN
    while curN != startN and curN != -1:
        curN = parents[curN]
        outPath = str(curN) + " --> " + outPath
    if str(curN) != startN:
        return "Not available"

    return outPath
```

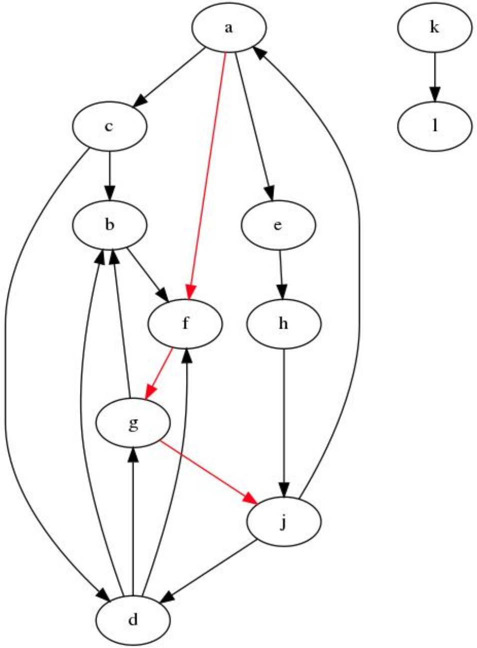


BFS: application. Shortest distance/Shortest path

printing the shortest path...

```python
def printPath(startN, endN, parents):
    outPath = str(endN)
    #this assumes all the nodes are in the
    #parents structure
    curN = endN
    while curN != startN and curN != -1:
        curN = parents[curN]
        outPath = str(curN) + " --> " + outPath
    if str(curN) != startN:
        return "Not available"

    return outPath
```

root or nodes not reached == -1



All parents: {'a': 'a', 'c': 'a', 'f': 'a', 'e': 'a', 'b': 'c', 'd': 'c', 'g': 'f', 'j': 'g', 'h': 'e', 'k': -1, 'l': -1}

```python
D, P = G2.get_distance('a')

print("Path from 'a' to 'j': {}".format(printPath('a','j', P)))
print("Path from 'a' to 'k': {}".format(printPath('a','k', P)))
```

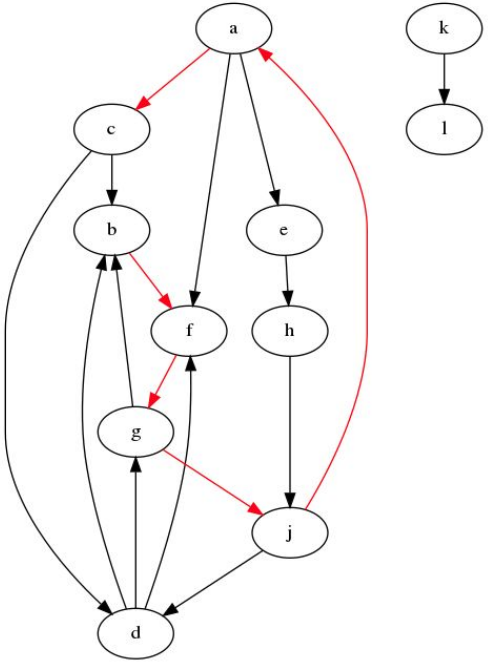
Path from 'a' to 'j': a --> f --> g --> j
Path from 'a' to 'k': Not available

BFS: application. Shortest distance/Shortest path

printing the shortest path...

```
def printPath(startN, endN, parents):
    outPath = str(endN)
    #this assumes all the nodes are in the
    #parents structure
    curN = endN
    while curN != startN and curN != -1:
        curN = parents[curN]
        outPath = str(curN) + " --> " + outPath
    if str(curN) != startN:
        return "Not available"

    return outPath
```

root or nodes not reached == -1



All parents: {'a': 'j', 'c': 'a', 'f': 'b', 'e': 'a', 'b': 'b', 'd': 'j', 'g': 'f', 'j': 'g', 'h': 'e', 'k': -1, 'l': -1}

```
D, P = G2.get_distance('b')
print("Distances from 'b': {}".format(D))
print("All parents: {}".format(P))
print("Path from 'b' to 'c': {}".format(printPath('b','c', P)))
```

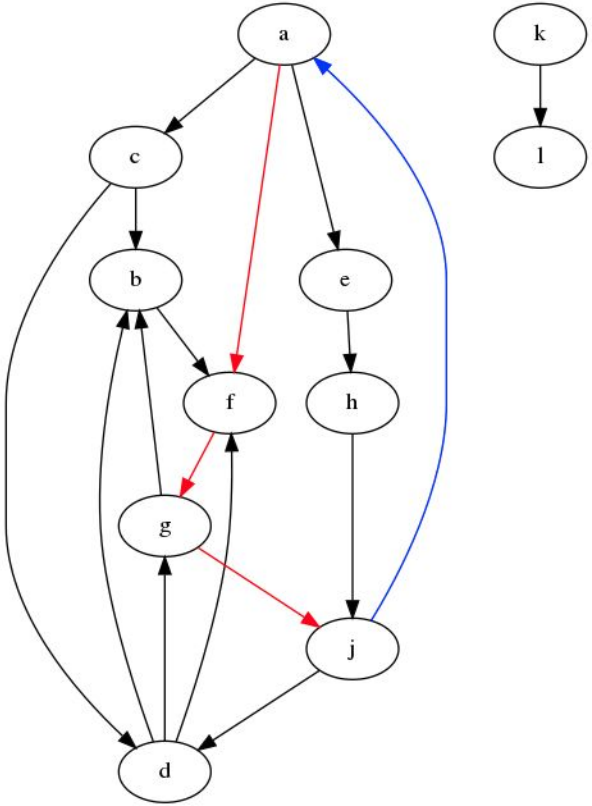
Path from 'b' to 'c': b --> f --> g --> j --> a --> c

Exercise

What if the shortest path between (a,j) is j➜ a???

```
def get_shortest_path(self, start, end):
    #your courtesy
    #returns [start, node,.., end]
    #if shortest path is start --> node --> ... --> end

    pass
```

Shortest path from 'a' to 'j': j --> a



Traversals: Depth First Search (DFS)

Depth-first search

- Often a subroutine of the solution of other problems
- Used to explore the entire graph, not just the nodes reachable from a single source (unlike BFS)

Output

- Instead of a tree, a depth-first forest $G_f = (V, E_f)$
- Contains a collection of depth-first trees

Data structure

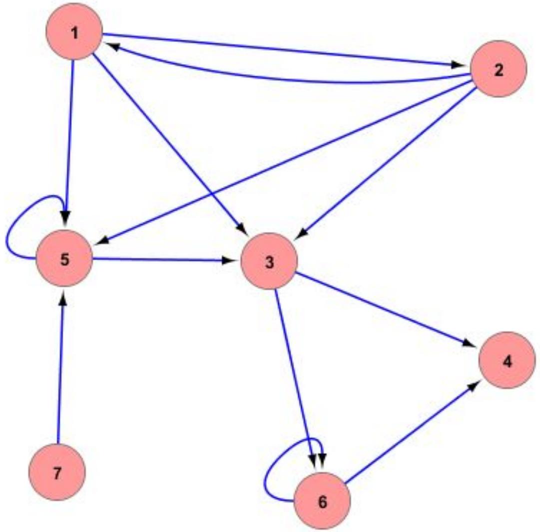
- Explicit Stack
- Or implicit stack, through recursion

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes
(**follow one path until it ends**)

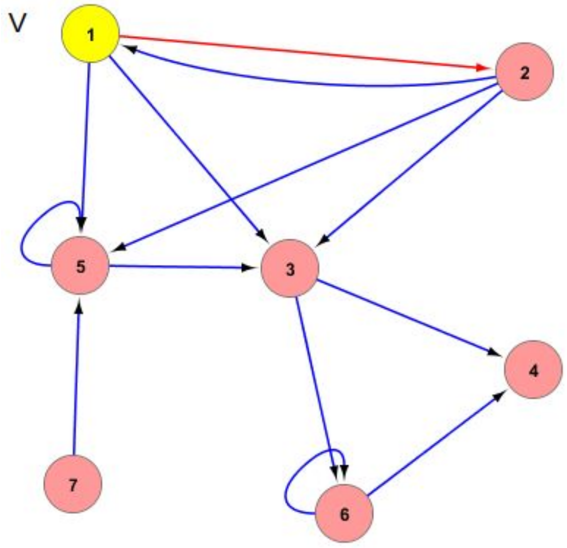


Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes
(**follow one path until it ends**)



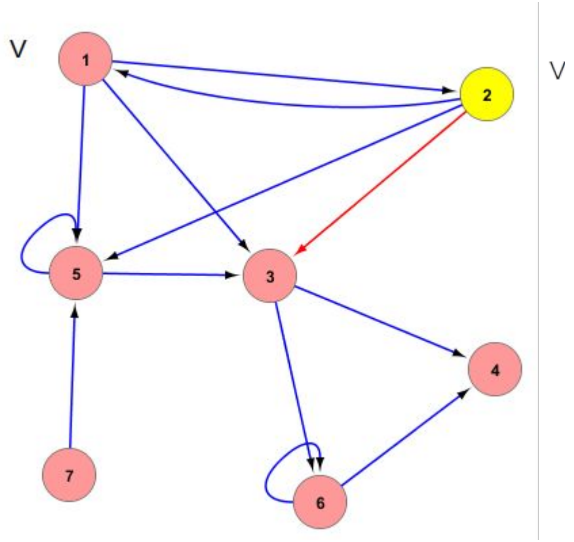
Execution stack:DFS(1)

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes
(**follow one path until it ends**)



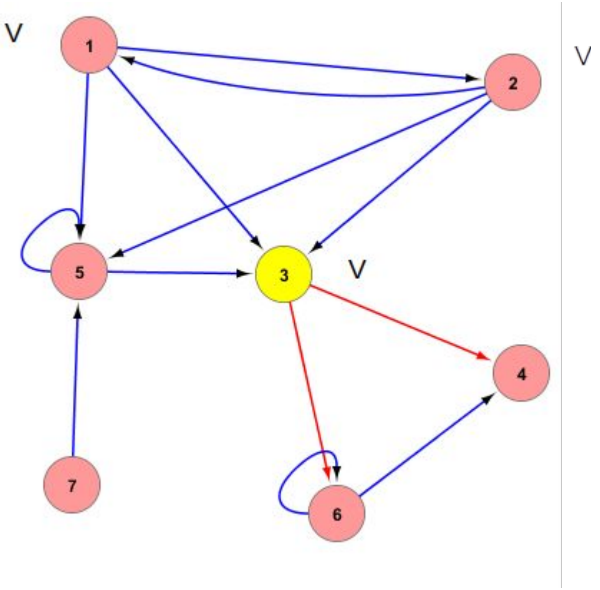
Execution stack:DFS(1, DFS(2))

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes (**follow one path until it ends**)



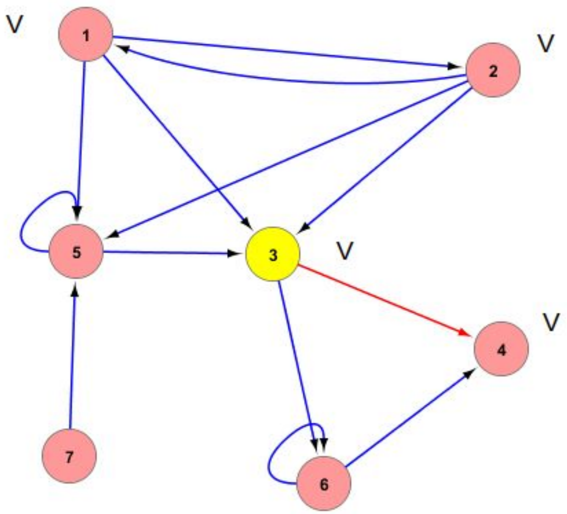
Execution stack:DFS(1, DFS(2, DFS(3)))

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes (**follow one path until it ends**)



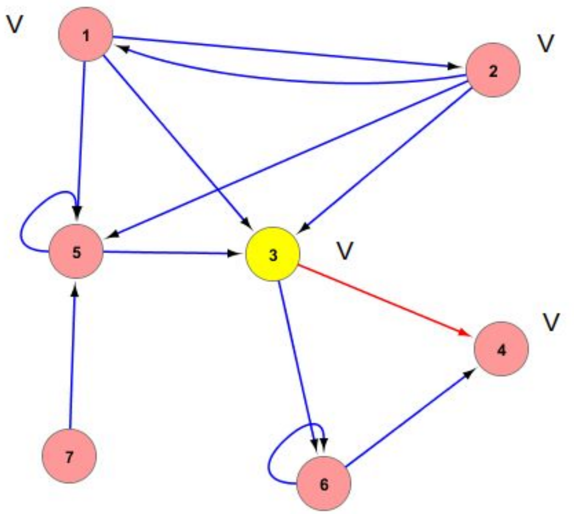
Execution stack:DFS(1, DFS(2, DFS(3, DFS(4))))

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes (**follow one path until it ends**)



Execution stack:DFS(1, DFS(2, DFS(3)))

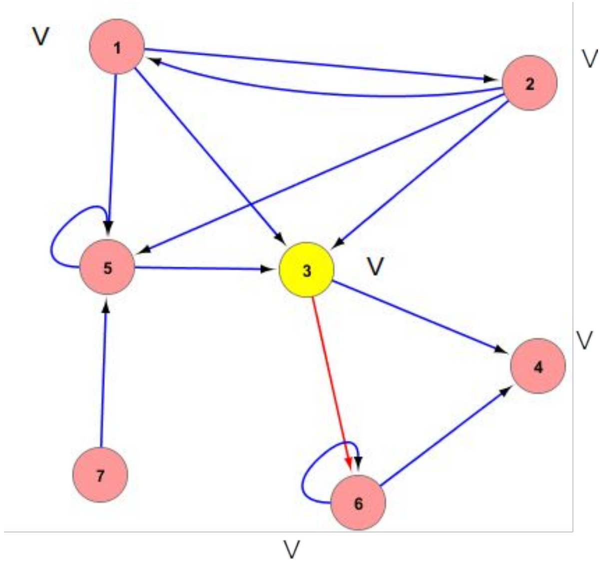
DFS(4): nothing to do. Done.

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes (**follow one path until it ends**)



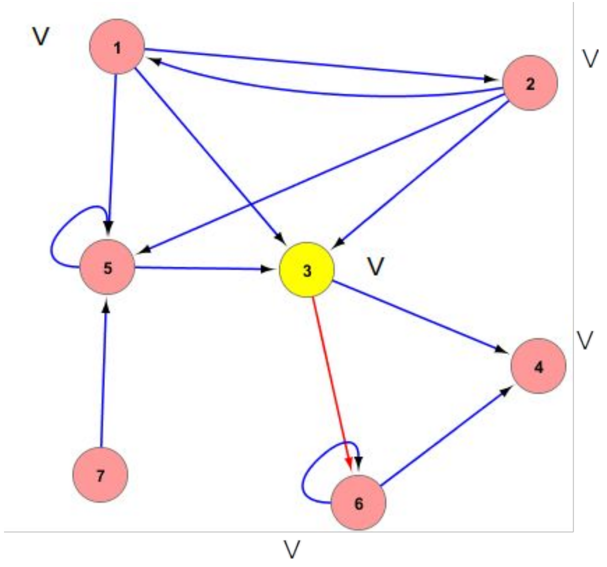
Execution stack:DFS(1, DFS(2, DFS(3, DFS(6))))

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes
(**follow one path until it ends**)



Execution stack:DFS(1, DFS(2, DFS(3))))

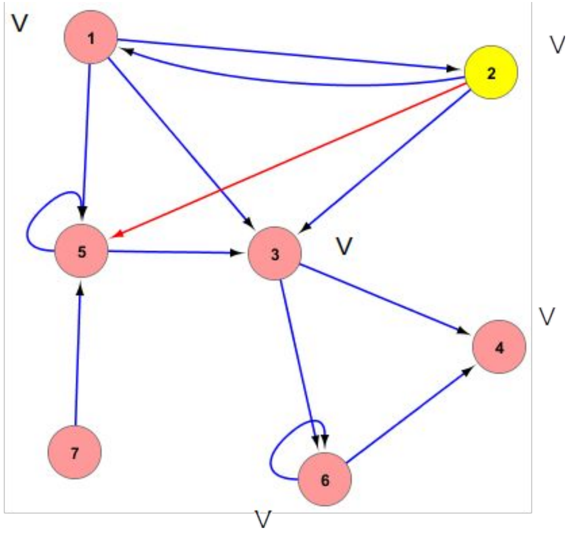
DFS(6): nothing to do. Done.

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes (**follow one path until it ends**)



Execution stack:DFS(1, DFS(2)))

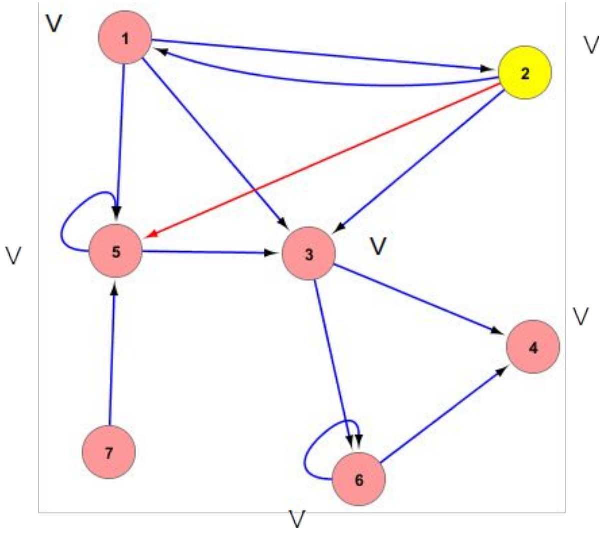
DFS(3): nothing to do. Done.

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes
(**follow one path until it ends**)



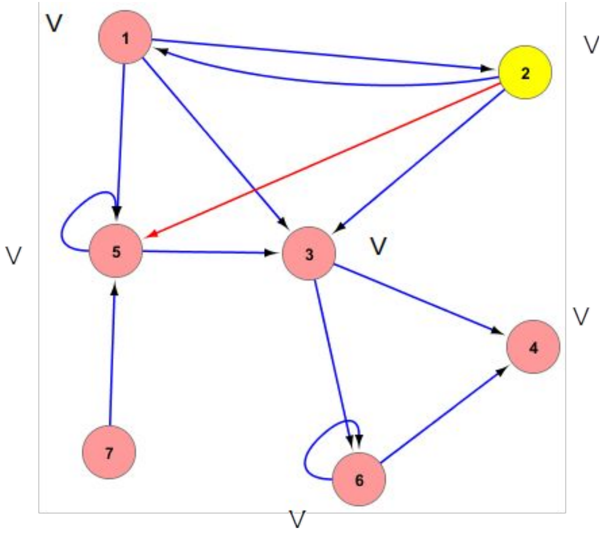
Execution stack:DFS(1, DFS(2, DFS(5))))

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes
(**follow one path until it ends**)



Execution stack:DFS(1, DFS(2))

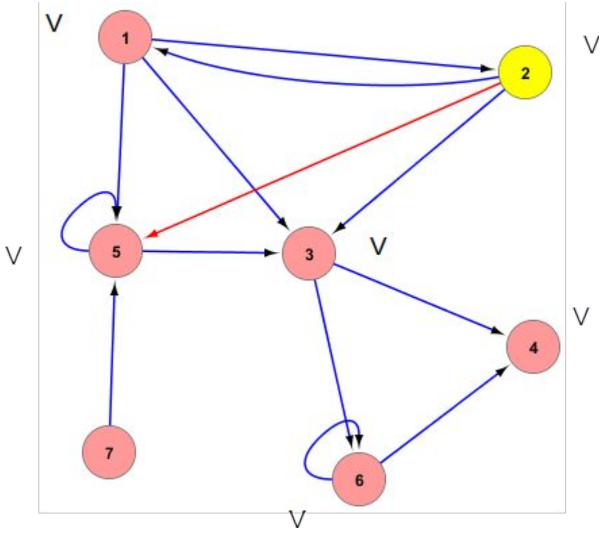
DFS(5): nothing to do. Done.

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes
(**follow one path until it ends**)



Execution stack:DFS(1)

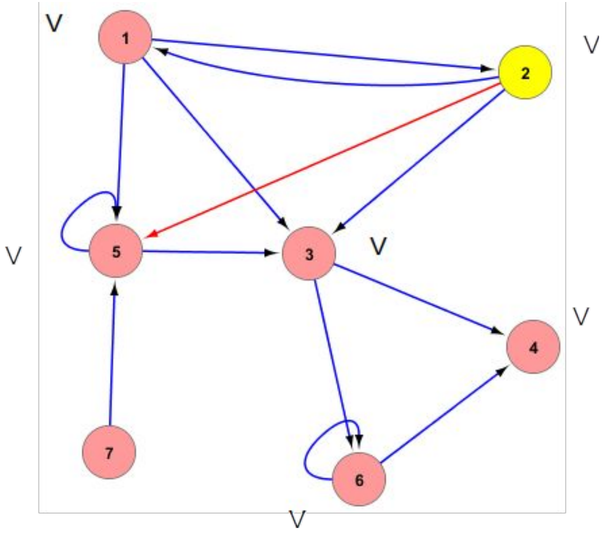
DFS(2): nothing to do. Done.

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes
(**follow one path until it ends**)



Execution stack: DONE!

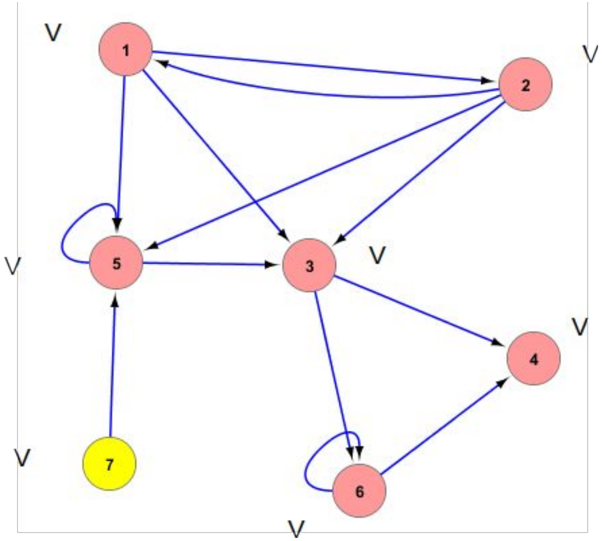
DFS(1): nothing to do. Done.

Traversals: Depth First Search (DFS)

Idea:

Visit the first node (**mark it as visited**)…

… then recursively all its children nodes (**follow one path until it ends**)



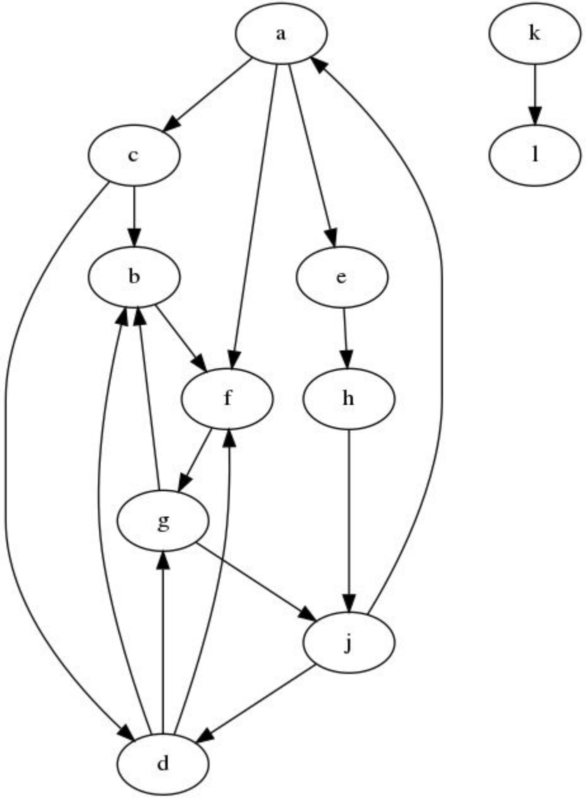
Execution stack: DFS(7)

Done.

Recursive Depth First Search (DFS)

```python
def DFS(self, node, visited):
    visited.add(node)
    ## visit node (preorder)
    print("visiting: {}".format(node))
    for u in self.adj(node):
        if u not in visited:
            self.DFS(u, visited)
    ##visit node (post-order)
```

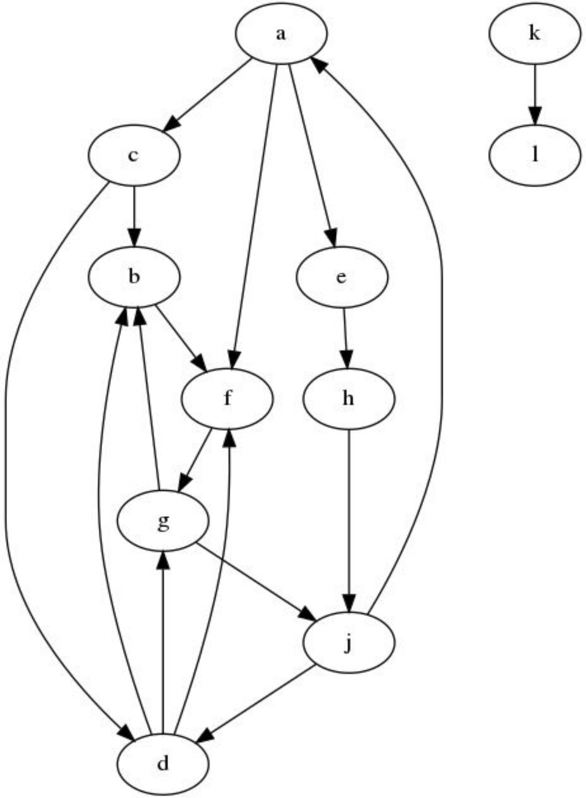
DFS from a:
visiting: a
visiting: c
visiting: b
visiting: f
visiting: g
visiting: j
visiting: d
visiting: e
visiting: h



Recursive Depth First Search (DFS)

```
def DFS(self, node, visited):
    visited.add(node)
    ## visit node (preorder)
    print("visiting: {}".format(node))
    for u in self.adj(node):
        if u not in visited:
            self.DFS(u, visited)
    ##visit node (post-order)
```

DFS from b:
visiting: b
visiting: f
visiting: g
visiting: j
visiting: a
visiting: c
visiting: d
visiting: e
visiting: h



Recursive Depth First Search (DFS)

- To execute a DFS based on recursive calls may be risky in very large graphs
- It is possible that the reached depth is larger than the size of the language stack
- In such cases, you should prefer a BFS or a DFS based on explicit stack

Stack size in Java

Platform	Default
Windows IA32	64 KB
Linux IA32	128 KB
Windows x86_64	128 KB
Linux x86_64	256 KB
Windows IA64	320 KB
Linux IA64	1024 KB (1 MB)
Solaris Sparc	512 KB

With recursive calls, “unclosed” calls are memorized in the stack and with big graphs this can cause a stack overflow error.

Iterative Depth First Search (DFS)

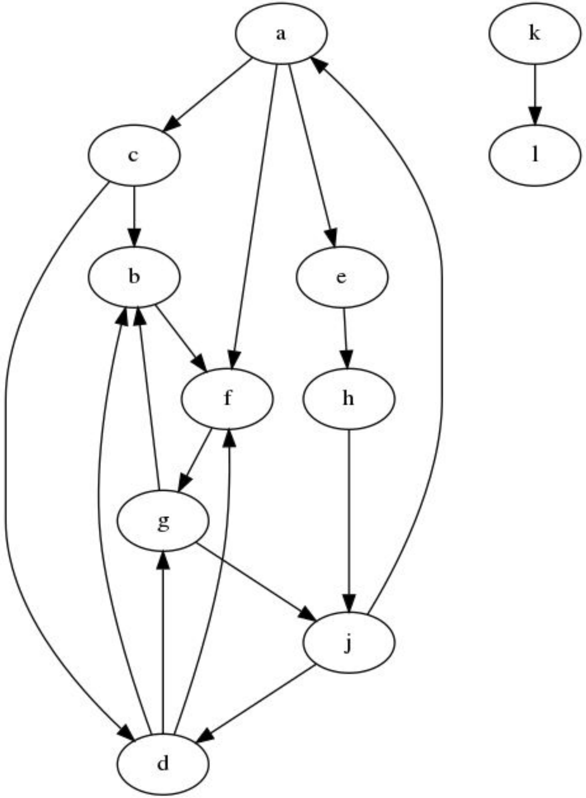
```python
def DFS(self, root):
    #stack implemented as deque
    S = deque()
    S.append(root)
    visited = set()
    while len(S) > 0:
        node = S.pop()
        if not node in visited:
            #visit node in preorder
            print("visiting {}".format(node))
            visited.add(node)
            for n in self.adj(node):
                #visit edge (node,n)
                S.append(n)
```

- A node can be inserted in the stack several times
- The check if a node has been already visited is done at the extraction, not when inserting
- Complexity $O(m+n)$
  - $O(m)$ edge visits
  - $O(m)$ insert, remove
  - $O(n)$ node visits

```python
print("DFS from a:")
G2.DFS('a')
print("DFS from b:")
G2.DFS('b')
```

DFS from a:
visiting a
visiting e
visiting h
visiting j
visiting d
visiting b
visiting f
visiting g
visiting c

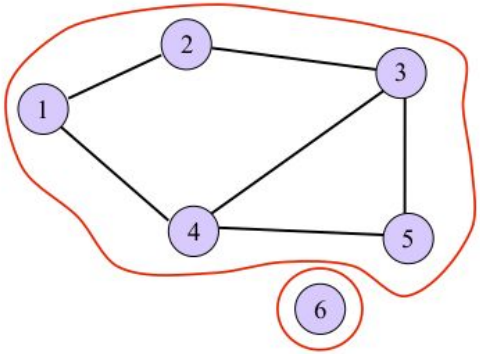
DFS from b:
visiting b
visiting f
visiting g
visiting j
visiting d
visiting a
visiting e
visiting h
visiting c



Connected graphs and components

Definitions

- An undirected graph $G = (V, E)$ is connected iff every node is reachable from every other node
- An undirected graph $G' = (V', E')$ is a connected component iff G' is a connected and maximal subgraph of G
- G' is a subgraph of G ($G' \subseteq G$) iff $V' \subseteq V$ and $E' \subseteq E$
- G' is maximal iff there is no other graph G'' of G such that G'' is connected and larger than G' (i.e. $G' \subseteq G'' \subseteq G$)



Connected components

Motivations

- Several algorithms that operate on graphs start by decomposing the graph into disconnected components
- The algorithm is then executed in each of the components
- The results are then composed back together

Definitions

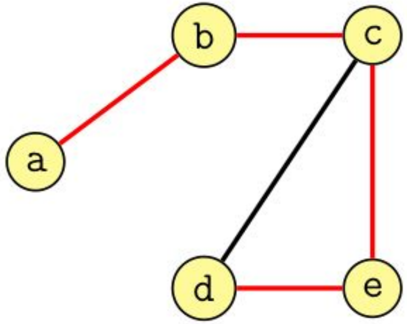
- Connected components (CC), defined on undirected graphs
- Strongly connected components (SCC), defined on directed graphs

Reachability

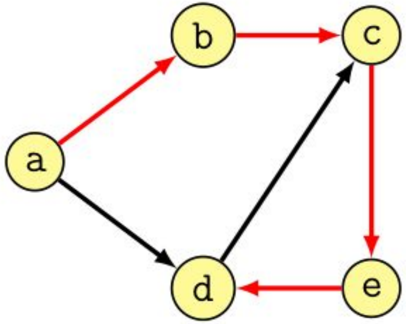
Reachable

A node v is reachable from a node u if there is at least one path from u to v.

Node d is reachable from node a and vice-versa



Node d is reachable from node A, but not vice-versa



Application of DFS

Problem

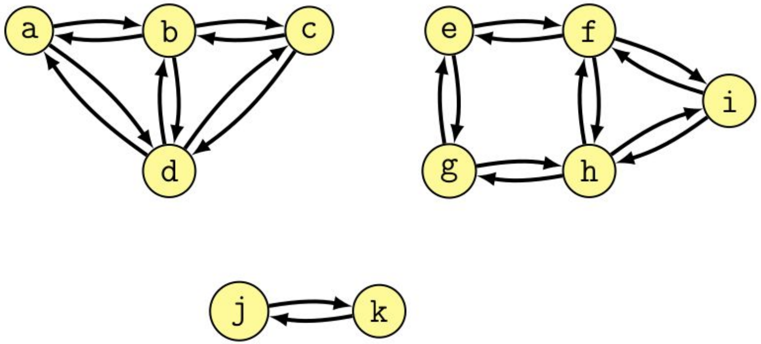
- To check whether an undirected graph is connected or not
- To identify its connected components

Solutions

- A graph is connected if, at the end of the DFS, all nodes have been marked
- If not, a single pass is not sufficient; the traversal must start again from an unmarked node, identifying a new component of the graph

Connected components

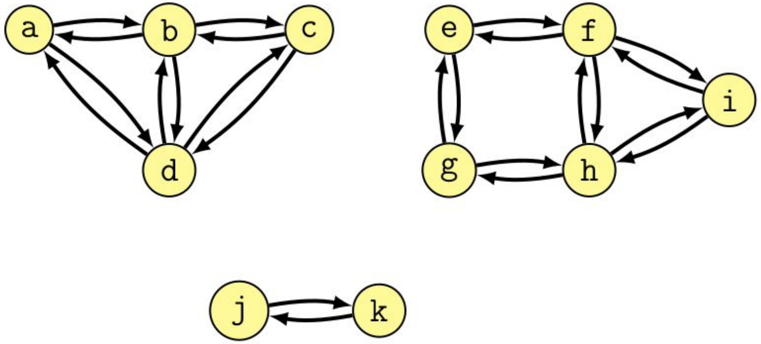
```python
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```



- ids is a list containing the component identifiers (it is also used as 'visited' structure)
- ids[u] is the identifier of the connected component to which u belongs

Connected components

```
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```

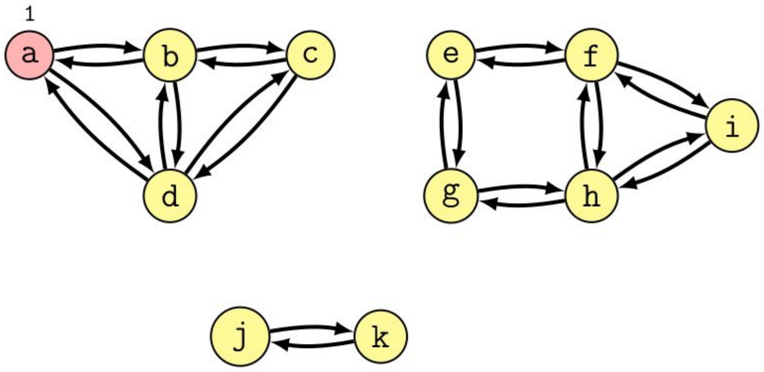



```
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Connected components

```python
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```

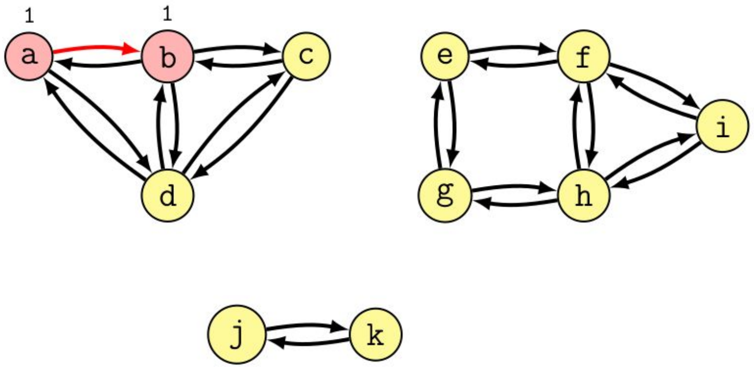



```python
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Connected components

```
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```



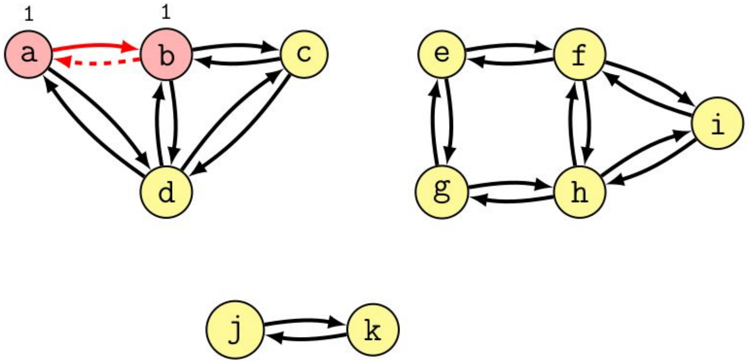

```
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Connected components

```
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```

ids is != 0

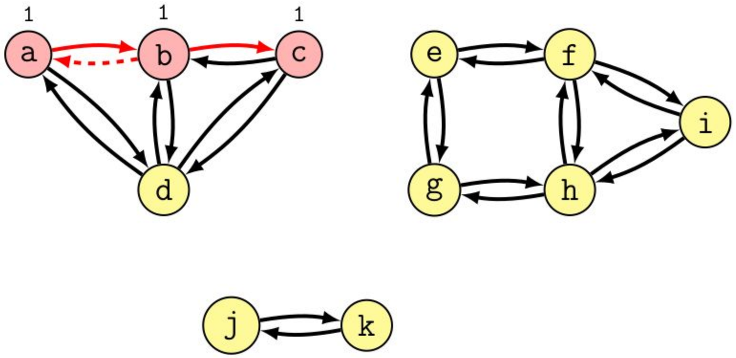



```
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Connected components

```
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```



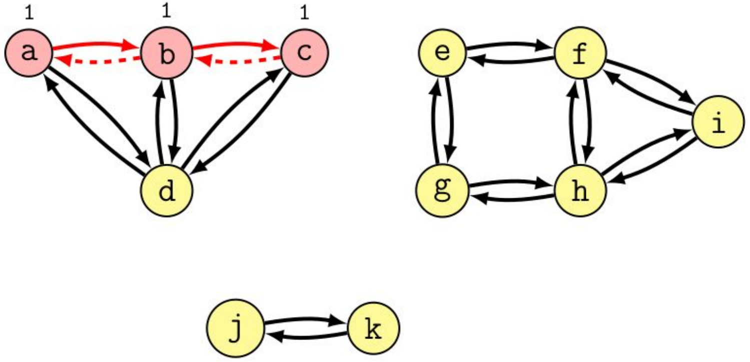

```
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Connected components

```python
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```

ids is != 0

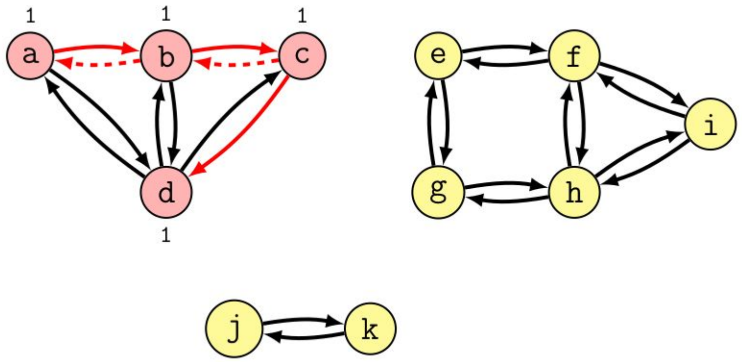



```python
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Connected components

```python
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```



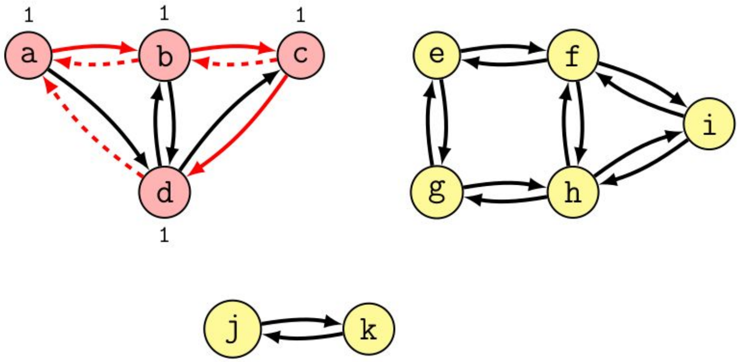

```python
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Connected components

```
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```

ids is != 0



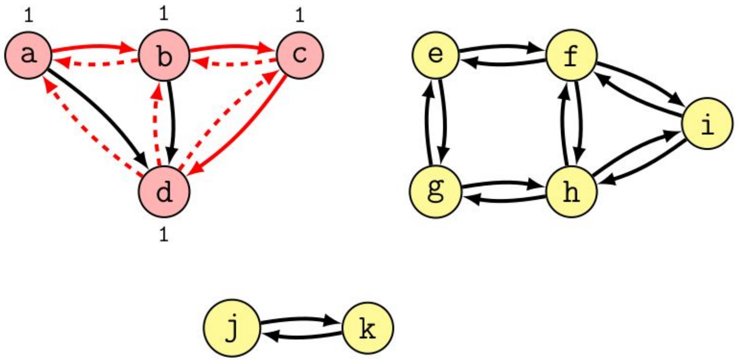

```
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Connected components

```python
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```

ids is != 0



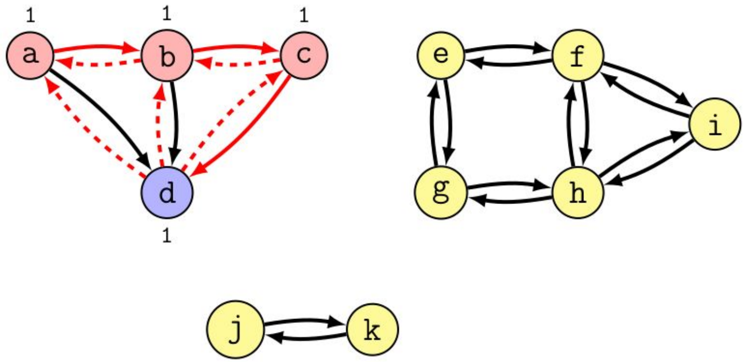

```python
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Connected components

```
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```

call on d
completed



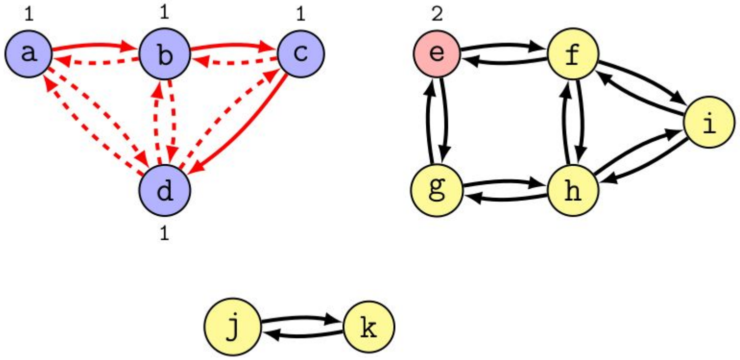

```
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Connected components

```
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```

call on c,b,a completed in the order
The algorithm tries to restart from b,c,d but nodes are visited...



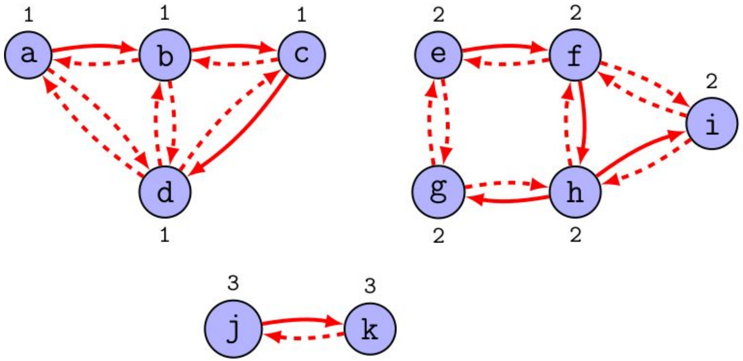
some steps later... component 1 is done, component 2 starts...

```
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Connected components

```
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```



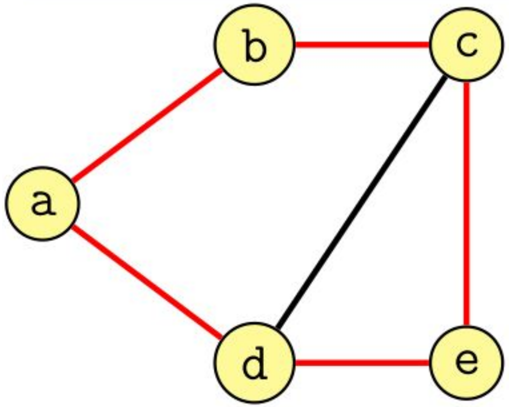

```
N, con_comp = cc(myG)
print("{} connected components:\n{}".format(N,con_comp))
```

3 connected components:
{'a': 1, 'b': 1, 'c': 1, 'd': 1, 'e': 2, 'g': 2, 'f': 2, 'h': 2, 'i': 2, 'j': 3, 'k': 3}

Definitions

Cycle

In a undirected graph $G = (V, E)$, a cycle C of length $k > 2$ is a sequence of nodes $u_0, u_1, \ldots, u_k$ such that $(u_i, u_{i+1} \in E)$ for $0 \leq i \leq k-1$ and $u_0 = u_k$.



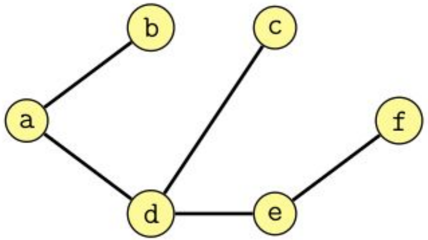
$k > 2$ is meant to exclude trivial cycles composed by edge pairs (u, v) and (v, u), which are everywhere in undirected graphs

Ignored, trivial cycle

Definitions

Acyclic graph

A undirected graph that does not contain cycles, is called acyclic.



Problem

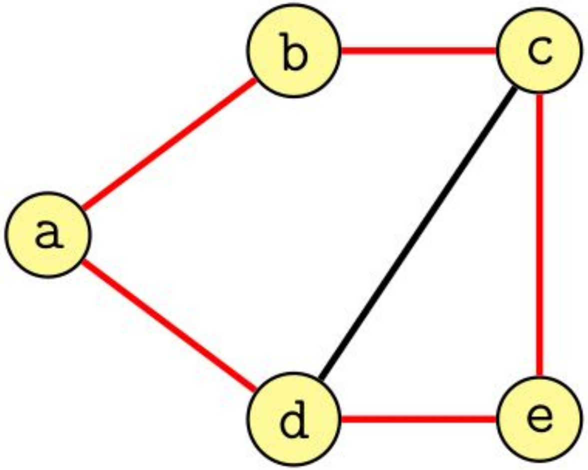
Given a undirected graph G, write an algorithm that returns **true** if G contains a cycle, **false** otherwise.

How would you solve the problem?

Idea: perform a DFS visit, if it finds a node already visited then there is a cycle

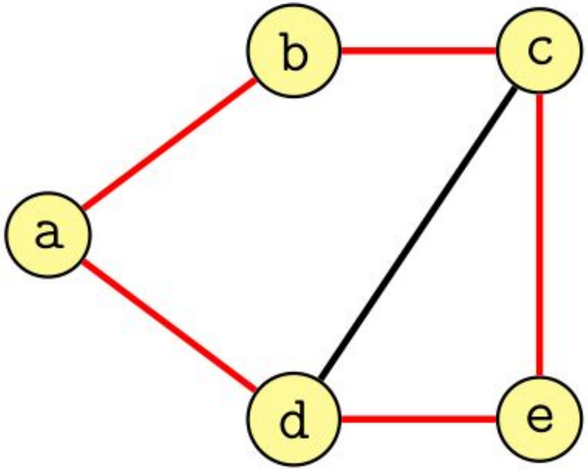
Cycle detection: undirected graph

```
def has_cycleRec(G, u, from_node, visited):
    visited.add(u)
    for v in G.adj(u):
        if v != from_node: #to avoid trivial cycles
            if v in visited:
                return True
            else:
                #continue with the visit to check
                #if there are cycles
                if has_cycleRec(G,v, u, visited):
                    return True
    return False


def has_cycle(G):
    visited = set()
    #I  am starting the visit from all nodes
    for node in G.node_iterator():
        if node not in visited:
            if has_cycleRec(G, node, None, visited):
                return True
    return False
```



Cycle detection: undirected graph

```python
def has_cycleRec(G, u, from_node, visited):
    visited.add(u)
    for v in G.adj(u):
        if v != from_node: #to avoid trivial cycles
            if v in visited:
                return True
            else:
                #continue with the visit to check
                #if there are cycles
                if has_cycleRec(G,v, u, visited):
                    return True
    return False


def has_cycle(G):
    visited = set()
    #I  am starting the visit from all nodes
    for node in G.node_iterator():
        if node not in visited:
            if has_cycleRec(G, node, None, visited):
                return True
    return False
```

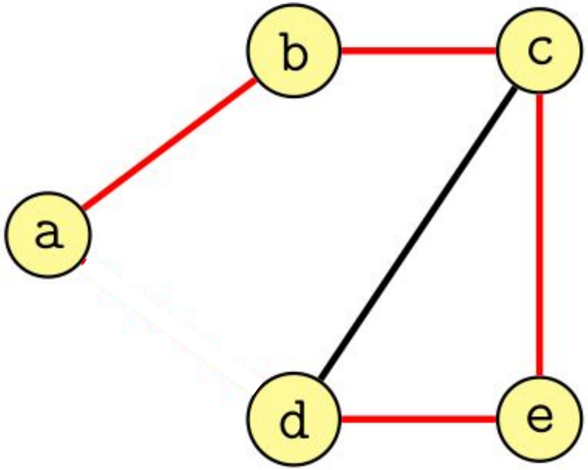



```python
for u, v in [('a', 'b'), ('b', 'a'), ('b', 'c'), ('c', 'b'), ('c', 'd'),
             ('d', 'c'), ('c','e'),('e','c'), ('d','a'),('a','d'), ('e','d'),
              ('d','e')]:
    myG.insert_edge(u,v)

print(has_cycle(myG))
```

True

Cycle detection: undirected graph

```python
def has_cycleRec(G, u, from_node, visited):
    visited.add(u)
    for v in G.adj(u):
        if v != from_node: #to avoid trivial cycles
            if v in visited:
                return True
            else:
                #continue with the visit to check
                #if there are cycles
                if has_cycleRec(G,v, u, visited):
                    return True
    return False


def has_cycle(G):
    visited = set()
    #I  am starting the visit from all nodes
    for node in G.node_iterator():
        if node not in visited:
            if has_cycleRec(G, node, None, visited):
                return True
    return False
```

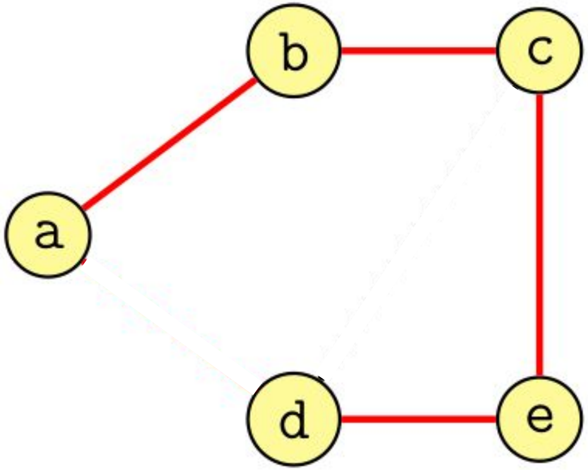



```python
myG  = Graph()
for u, v in [('a', 'b'), ('b', 'a'), ('b', 'c'), ('c', 'b'), ('c', 'd'),
             ('d', 'c'), ('c','e'),('e','c'), ('e','d'),
              ('d','e')]:
    myG.insert_edge(u,v)

print(has_cycle(myG))
```

True

Cycle detection: undirected graph

```
def has_cycleRec(G, u, from_node, visited):
    visited.add(u)
    for v in G.adj(u):
        if v != from_node: #to avoid trivial cycles
            if v in visited:
                return True
            else:
                #continue with the visit to check
                #if there are cycles
                if has_cycleRec(G,v, u, visited):
                    return True
    return False


def has_cycle(G):
    visited = set()
    #I  am starting the visit from all nodes
    for node in G.node_iterator():
        if node not in visited:
            if has_cycleRec(G, node, None, visited):
                return True
    return False
```



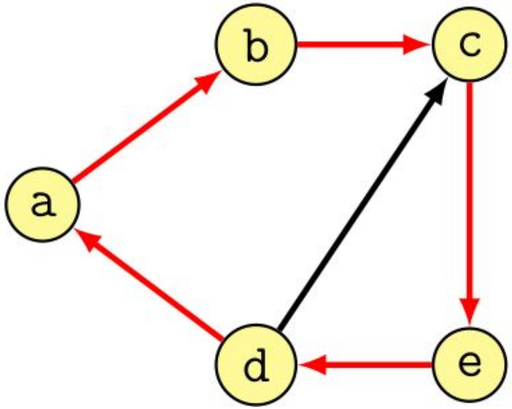

```
myG  = Graph()
for u, v in [('a', 'b'), ('b', 'a'), ('b', 'c'), ('c', 'b'),
             ('c','e'),('e','c'), ('e','d'),
             ('d','e')]:
    myG.insert_edge(u,v)

print(has_cycle(myG))
```

False

Cycle detection: directed graph

Cycle

In a directed graph $G = (V, E)$, a cycle C of length $k \geq 2$ is a sequence of nodes $u_0, u_1, \ldots, u_k$ such that $(u_i, u_{i+1} \in E)$ for $0 \leq i \leq k-1$ and $u_0 = u_k$.



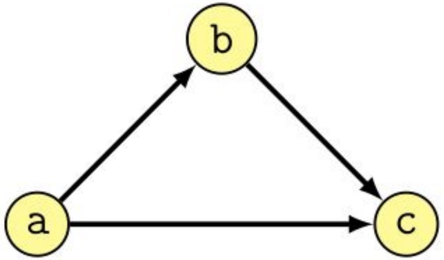
Example: a, b, c, e, d, a is a cycle of length 5

Note: a cycle is called simple if all its nodes are distinct (excluding the first and the last ones)

Directed acyclic graph (DAG)

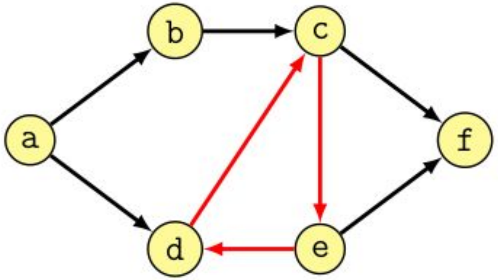
DAG

A directed acyclic graph (DAG) is a directed graph that does not contain cycles.



Cyclic graph

A graph containing a cycle is called cyclic



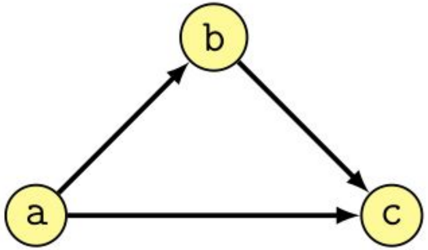
Cycle detection

Problem

Given a directed graph G, write an algorithm that returns **true** if G contains a cycle, **false** otherwise.

Problem

Can you draw a directed graph such that the algorithm we have seen before does not return the correct answer?



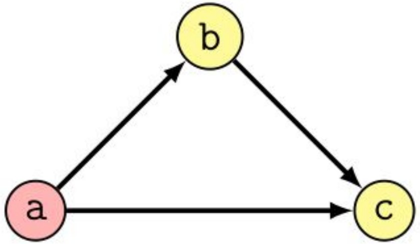
Cycle detection

Problem

Given a directed graph G, write an algorithm that returns **true** if G contains a cycle, **false** otherwise.

Problem

Can you draw a directed graph such that the algorithm we have seen before does not return the correct answer?



visit a

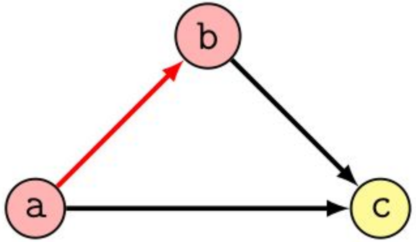
Cycle detection

Problem

Given a directed graph G, write an algorithm that returns **true** if G contains a cycle, **false** otherwise.

Problem

Can you draw a directed graph such that the algorithm we have seen before does not return the correct answer?



visit b

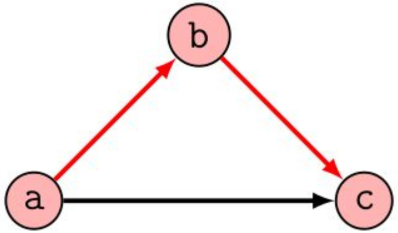
Cycle detection

Problem

Given a directed graph G, write an algorithm that returns **true** if G contains a cycle, **false** otherwise.

Problem

Can you draw a directed graph such that the algorithm we have seen before does not return the correct answer?



visit c

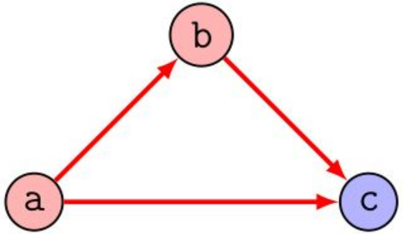
Cycle detection

Problem

Given a directed graph G, write an algorithm that returns **true** if G contains a cycle, **false** otherwise.

Problem

Can you draw a directed graph such that the algorithm we have seen before does not return the correct answer?



back from a to c
→ **cycle: wrong answer**

Edge classification

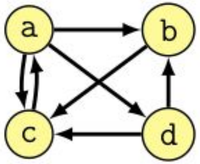
DFS Spanning Tree

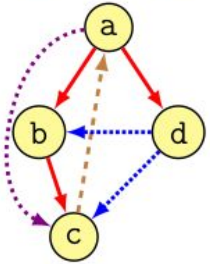
Whenever an edge connecting a marked node to an unmarked one, it is inserted into a tree T

Every edge (u, v) not included in T belongs to one of three categories

edges part of the DFS visit

- (u, v) is a forward edge iff v is a descendent of u in T
- (u, v) is a back edge iff v is an ancestor of u in T
- Otherwise, $(u, v)$is a cross edge





Edge classification

```python
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```

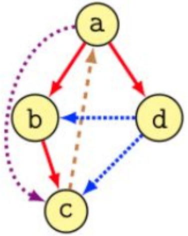
clock is increased by one at each operation

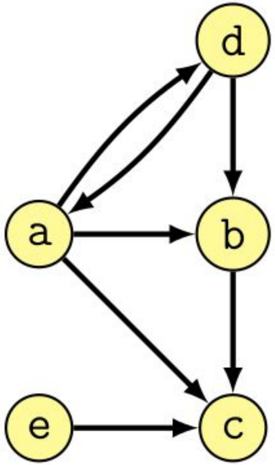
perform a DFS visit
if dt[v] == 0 → equals to v NOT visited

increase the time and set the finish time of node

```python
dt = dict()
df = dict()
for node in G.node_iterator():
    dt[node] = 0
    df[node] = 0
```

DFS edge
Forward edge
Back edge
Cross edge

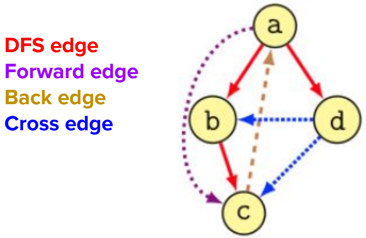




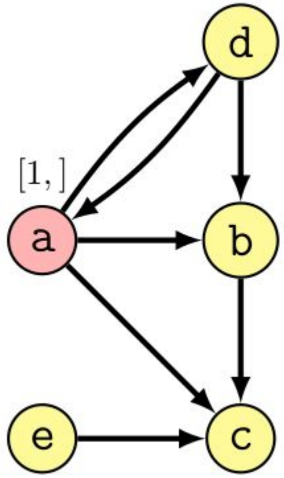

```python
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification

```
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```



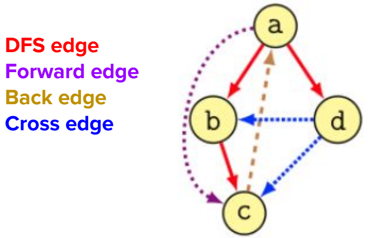
Start time a: 1



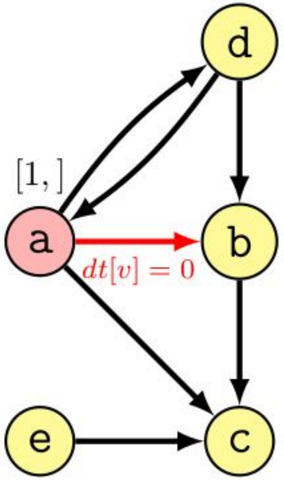

```
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification

```python
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```



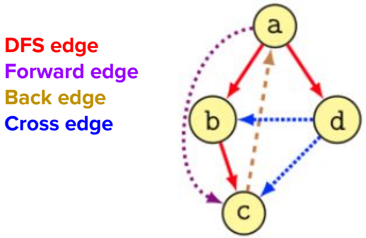
Start time a: 1
DFS edge: a --> b



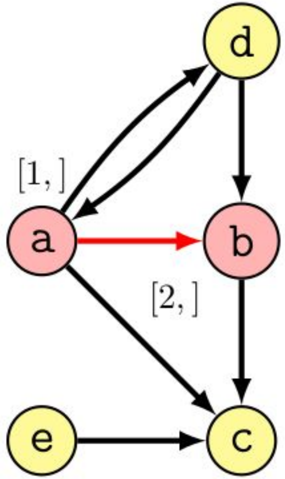

```python
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification

```
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```



Start time a: 1
DFS edge: a --> b
Start time b: 2

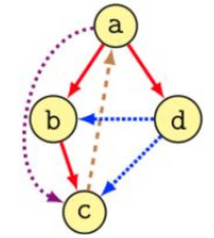



```
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

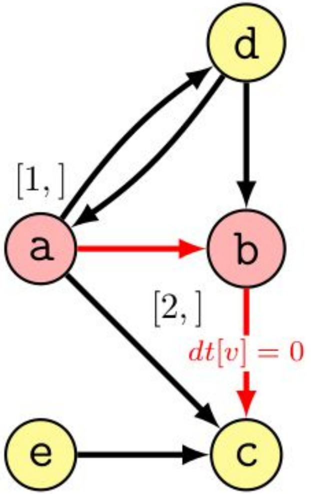
Edge classification

```
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```

DFS edge
Forward edge
Back edge
Cross edge



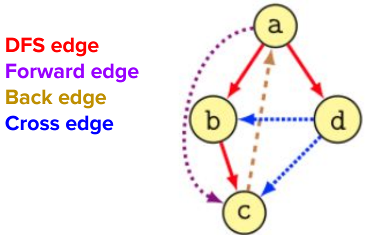
Start time a: 1
    DFS edge: a --> b
Start time b: 2
    DFS edge: b --> c



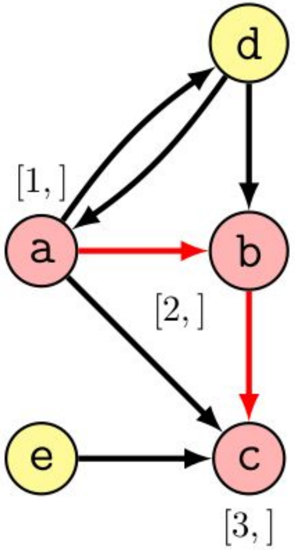

```
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification

```python
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```




```
Start time a: 1
        DFS edge: a --> b
Start time b: 2
        DFS edge: b --> c
Start time c: 3
```

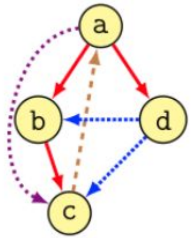



```python
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

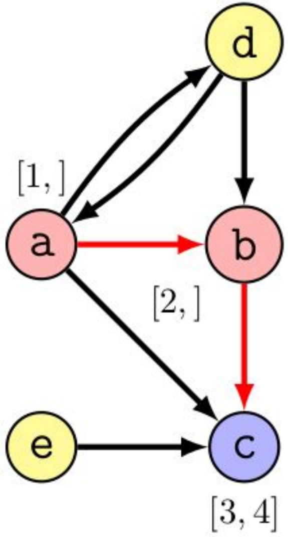
Edge classification

```python
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```

DFS edge
Forward edge
Back edge
Cross edge



Start time a: 1
    DFS edge: a --> b
Start time b: 2
    DFS edge: b --> c
Start time c: 3
Finish time c: 4

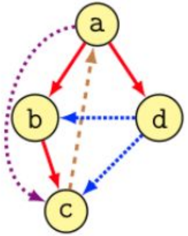



```python
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

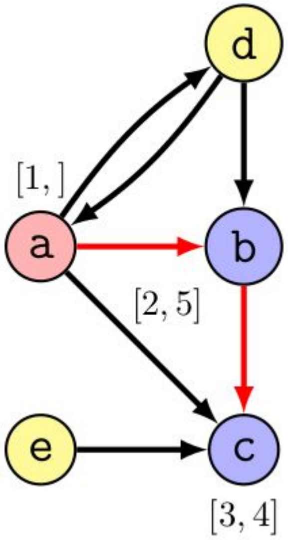
Edge classification

```
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```

DFS edge
Forward edge
Back edge
Cross edge



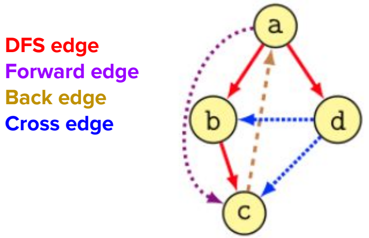

```
Start time a: 1
        DFS edge: a --> b
Start time b: 2
        DFS edge: b --> c
Start time c: 3
Finish time c: 4
Finish time b: 5
```



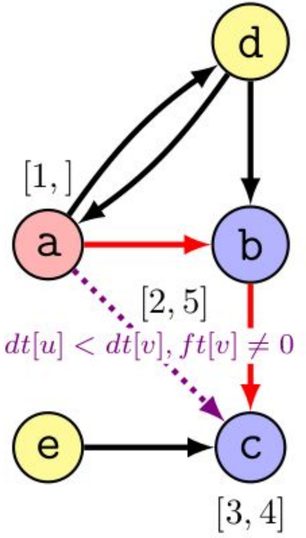

```
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification

```python
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```



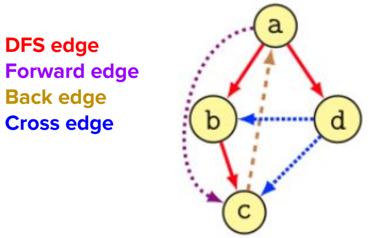
Start time a: 1
    DFS edge: a --> b
Start time b: 2
    DFS edge: b --> c
Start time c: 3
Finish time c: 4
Finish time b: 5
    Forward edge: a--> c



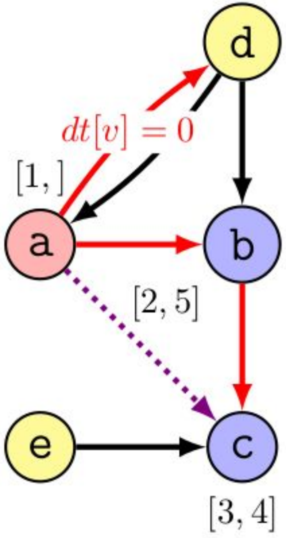

```python
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification

```python
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```



Start time a: 1
    DFS edge: a --> b
Start time b: 2
    DFS edge: b --> c
Start time c: 3
Finish time c: 4
Finish time b: 5
    Forward edge: a--> c
    DFS edge: a --> d




```python
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

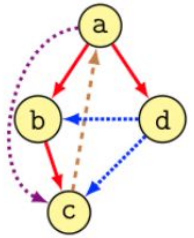
Edge classification

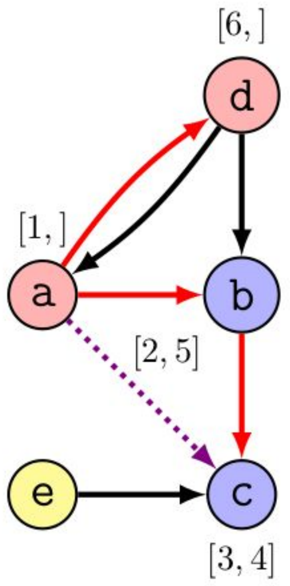
```python
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```

Start time a: 1
    DFS edge: a --> b
Start time b: 2
    DFS edge: b --> c
Start time c: 3
Finish time c: 4
Finish time b: 5
    Forward edge: a--> c
    DFS edge: a --> d
Start time d: 6

```python
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

DFS edge
Forward edge
Back edge
Cross edge



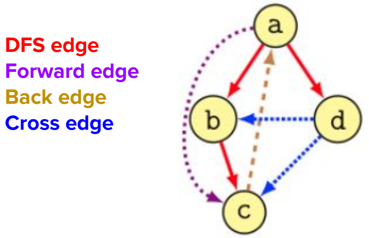


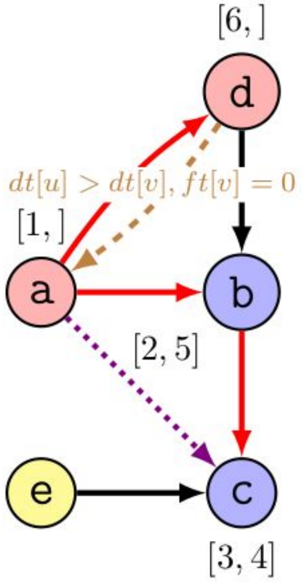
Edge classification

```
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```

Start time a: 1
    DFS edge: a --> b
Start time b: 2
    DFS edge: b --> c
Start time c: 3
Finish time c: 4
Finish time b: 5
    Forward edge: a--> c
    DFS edge: a --> d
Start time d: 6
    Back edge: d--> a

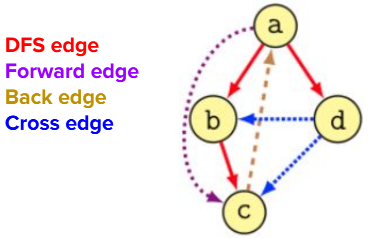
```
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```



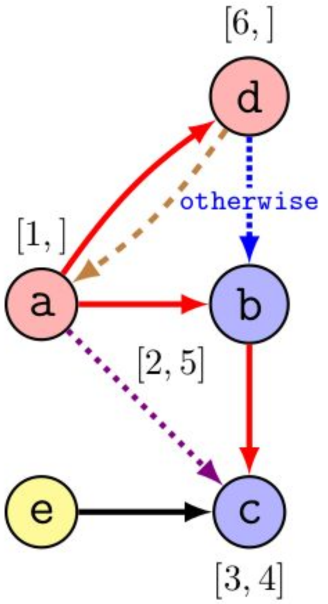


Edge classification

```python
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```



Start time a: 1
DFS edge: a --> b
Start time b: 2
DFS edge: b --> c
Start time c: 3
Finish time c: 4
Finish time b: 5
Forward edge: a--> c
DFS edge: a --> d
Start time d: 6
Back edge: d--> a
Cross edge: d --> b

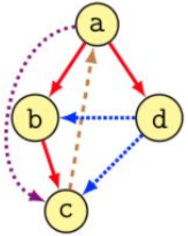


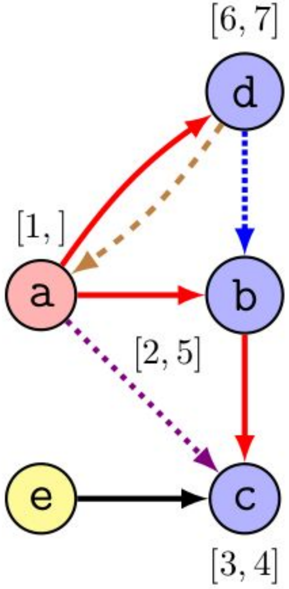

```python
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification

```python
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```

DFS edge
Forward edge
Back edge
Cross edge



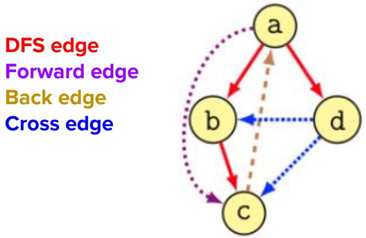



```
Start time a: 1
        DFS edge: a --> b
Start time b: 2
        DFS edge: b --> c
Start time c: 3
Finish time c: 4
Finish time b: 5
        Forward edge: a--> c
        DFS edge: a --> d
Start time d: 6
        Back edge: d--> a
        Cross edge: d --> b
Finish time d: 7
```

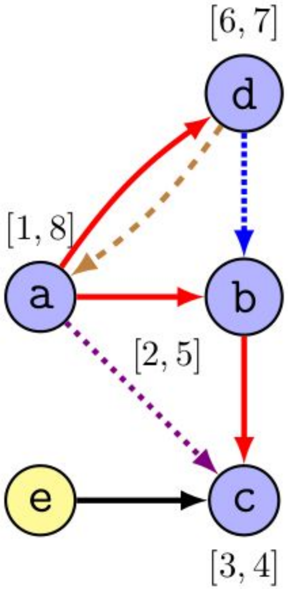
```python
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification

```
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```



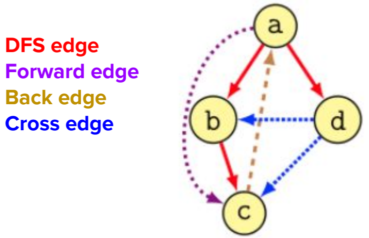
Start time a: 1
DFS edge: a --> b
Start time b: 2
DFS edge: b --> c
Start time c: 3
Finish time c: 4
Finish time b: 5
Forward edge: a--> c
DFS edge: a --> d
Start time d: 6
Back edge: d--> a
Cross edge: d --> b
Finish time d: 7
Finish time a: 8



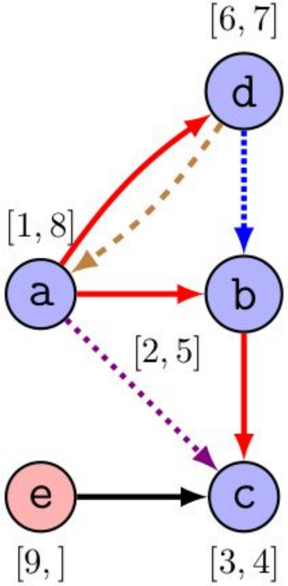

```
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification

```python
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```



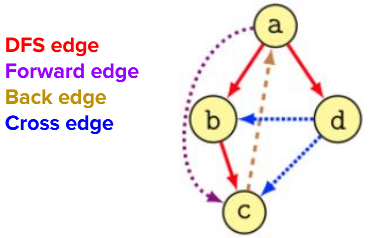
Start time a: 1
DFS edge: a --> b
Start time b: 2
DFS edge: b --> c
Start time c: 3
Finish time c: 4
Finish time b: 5
Forward edge: a--> c
DFS edge: a --> d
Start time d: 6
Back edge: d--> a
Cross edge: d --> b
Finish time d: 7
Finish time a: 8
Start time e: 9



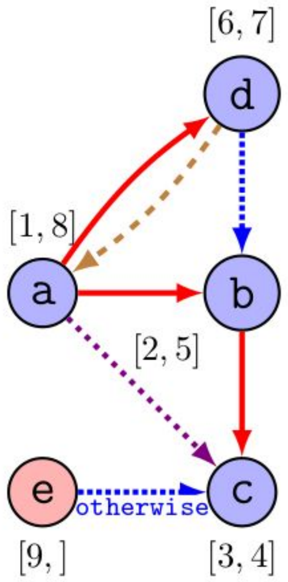

```python
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification

```
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```



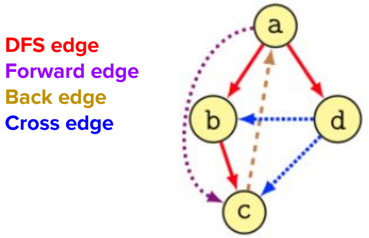
Start time a: 1
    DFS edge: a --> b
Start time b: 2
    DFS edge: b --> c
Start time c: 3
Finish time c: 4
Finish time b: 5
    Forward edge: a--> c
    DFS edge: a --> d
Start time d: 6
    Back edge: d--> a
    Cross edge: d --> b
Finish time d: 7
Finish time a: 8
Start time e: 9
    Cross edge: e --> c



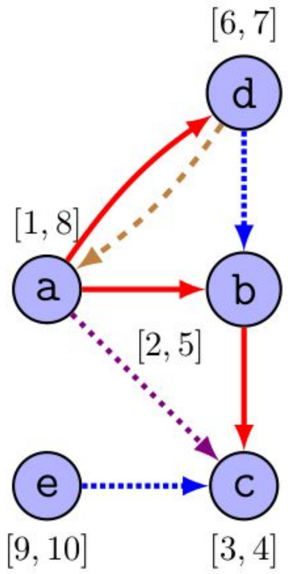

```
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification

```python
clock = 0

def dfs_schema(G, node, dt, ft):
    #clock: visit time (global variable)
    #dt: discovery time
    #ft: finish time
    global clock

    clock += 1
    dt[node] = clock
    print("Start time {}: {}".format(node, clock))

    for v in G.adj(node):
        if dt[v] == 0:
            #DFS VISIT edge
            #visit the edge (node,v)
            print("\tDFS edge: {} --> {}".format(node, v))
            dfs_schema(G,v, dt, ft)
        elif dt[node] > dt[v] and ft[v] == 0:
            #BACK EDGE
            #visit the back edge (node,v)
            print("\tBack edge: {}--> {}".format(node,v))
        elif dt[node] < dt[v] and ft[v] != 0:
            #FORWARD EDGE
            #visit the forward edge (node,v)
            print("\tForward edge: {}--> {}".format(node,v))
        else:
            #CROSS EDGE
            print("\tCross edge: {} --> {}".format(node,v))
    clock += 1
    ft[node] = clock
    print("Finish time {}: {}".format(node,clock))
    return dt,ft
```

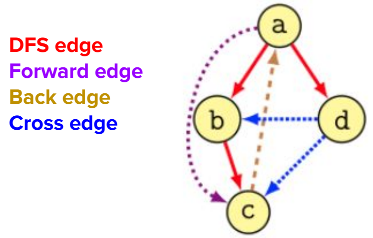



```
Start time a: 1
        DFS edge: a --> b
Start time b: 2
        DFS edge: b --> c
Start time c: 3
Finish time c: 4
Finish time b: 5
        Forward edge: a--> c
        DFS edge: a --> d
Start time d: 6
        Back edge: d--> a
        Cross edge: d --> b
Finish time d: 7
Finish time a: 8
Start time e: 9
        Cross edge: e --> c
Finish time e: 10
Discovery times:{'a': 1, 'b': 2, 'c': 3, 'd': 6, 'e': 9}
Finish times: {'a': 8, 'b': 5, 'c': 4, 'd': 7, 'e': 10}
```




```python
s,e = dfs_schema(G,'a',  dt, df)
s,e = dfs_schema(G,'e',  dt, df)
print("Discovery times:{}".format(s))
print("Finish times: {}".format(e))
```

Edge classification



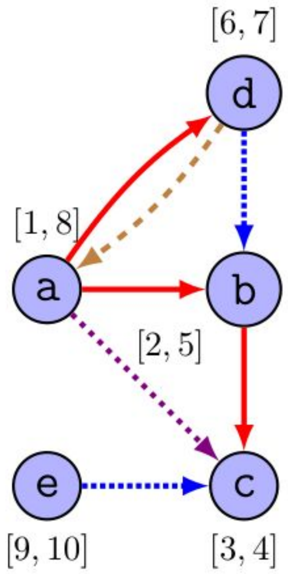
Why are we classifying edges?

We can prove properties on the type of edges and use these properties to build better algorithms

Theorem

In each DFS visit of a graph $G = (V, E)$, for each pair of nodes $u, v \in V$, only one of the following conditions is true:

- The intervals $[dt[u], ft[u]]$ e $[dt[v], ft[v]]$ are non-overlapping; u, v are not descendant of each other in the DF forest
- Interval $[dt[u], ft[u]]$ is completely contained in $[dt[v], ft[v]]$; u is descendant of v in a DF tree
- Interval $[dt[v], ft[v]]$ is completely contained in $[dt[u], ft[u]]$; v is descendant of u in a DF tree

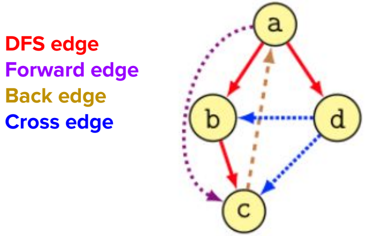


NOTE in the DFS visit:

[1,8] completely contains [2,5] → B descends from A
[1,8] completely contains [3,4] → C descends from A
[9,10] does not overlap [2,5], [6,7] → E-B E-D are not descendans

Intervals describe the relationship between nodes

Cycle detection

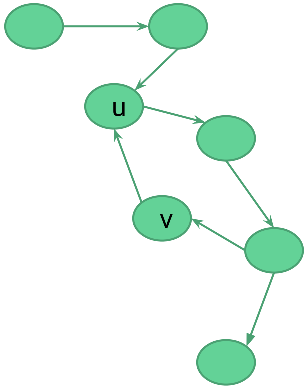


Theorem

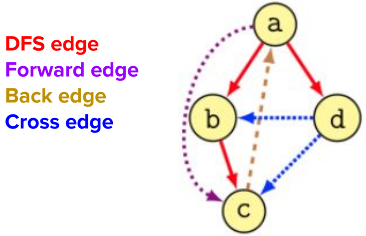
A graph G contains a cycle if a back edge is found when a DFS is performed on G.

Informal proof

- **if**: If there is a cycle, let u be the first node of it that is visited. Given that u belongs to the cycle, there is an edge (v, u) in the cycle. Given that v belongs to the cycle, there is a path from u to v. So (v, u) is a back edge.
- **only if**: if there is a back edge (u, v), where v is an ancestor of u, then there is a path from v to u and an edge from u to v, thus there is a cycle.



Cycle detection

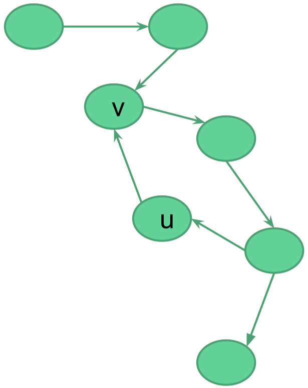


Theorem

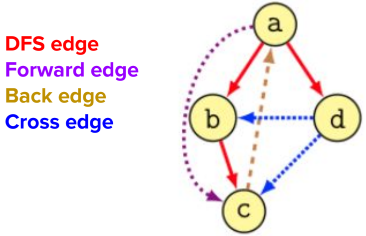
A graph G contains a cycle if a back edge is found when a DFS is performed on G.

Informal proof

- **if**: If there is a cycle, let u be the first node of it that is visited. Given that u belongs to the cycle, there is an edge (v, u) in the cycle. Given that v belongs to the cycle, there is a path from u to v. So (v, u) is a back edge.
- **only if**: if there is a back edge (u, v), where v is an ancestor of u, then there is a path from v to u and an edge from u to v, thus there is a cycle.



Cycle detection



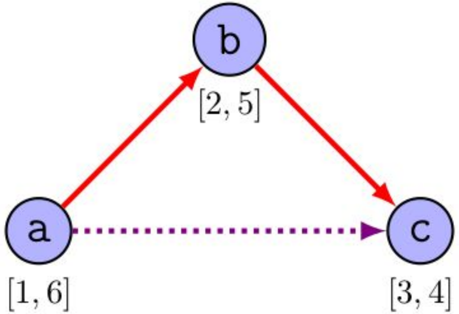
Theorem

A graph G contains a cycle if a back edge is found when a DFS is performed on G.

Informal proof

- **if**: If there is a cycle, let u be the first node of it that is visited. Given that u belongs to the cycle, there is an edge (v, u) in the cycle. Given that v belongs to the cycle, there is a path from u to v. So (u, v) is a back edge.
- **only if**: if there is a back edge (u, v), where v is an ancestor of u, then there is a path from v to u and an edge from u to v, thus there is a cycle.

NO Cycle!



Tree edge	$dt[v] == 0$
Back edge:	$dt[u] > dt[v]$ **and** $ft[v] = 0$
Forward edge:	$dt[u] < dt[v]$ **and** $ft[v] \neq 0$
Cross edge:	otherwise

Cycle detection

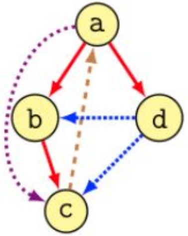
Theorem

A graph G contains a cycle if a back edge is found when a DFS is performed on G.

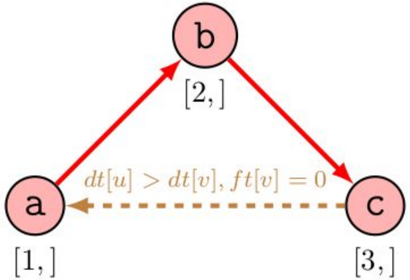
Informal proof

- **if**: If there is a cycle, let u be the first node of it that is visited. Given that u belongs to the cycle, there is an edge (v, u) in the cycle. Given that v belongs to the cycle, there is a path from u to v. So (u, v) is a back edge.
- **only if**: if there is a back edge (u, v), where v is an ancestor of u, then there is a path from v to u and an edge from u to v, thus there is a cycle.

DFS edge
Forward edge
Back edge
Cross edge

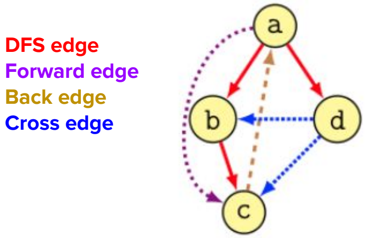


Cycle!



Tree edge $dt[v] == 0$
Back edge: $dt[u] > dt[v]$ **and** $ft[v] = 0$
Forward edge: $dt[u] < dt[v]$ **and** $ft[v] \neq 0$
Cross edge: otherwise

Cycle detection: the code

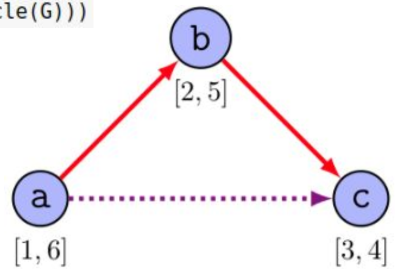



```python
def detect_cycle(G):
    dt = dict()
    ft = dict()
    global clock

    def has_cycle(G, node, dt, ft):
        #clock: visit time (global variable)
        #dt: discovery time
        #ft: finish time
        global clock

        clock += 1
        dt[node] = clock
        for v in G.adj(node):
            if dt[v] == 0:
                #DFS VISIT edge
                if has_cycle(G,v, dt, ft):
                    return True
            elif dt[node] > dt[v] and ft[v] == 0:
                #BACK EDGE
                #CYCLE FOUND!!!!
                print("Back edge: {} --> {}".format(node,v))
                return True
            ## Note we are not interested
            ## in forward and cross edges

        clock += 1
        ft[node] = clock
        return False


    for node in G.node_iterator():
        dt[node] = 0
        ft[node] = 0
    clock = 1
    for u in G.node_iterator():
        if ft[u] == 0:
            if has_cycle(G,u, dt, ft):
                return True
    return False
```

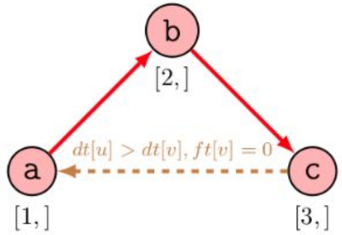
simplified version of the code seen before.
We just care about forward and back edges

```python
print("Does G have a cycle? {}".format(detect_cycle(G)))
```

Does G have a cycle? False

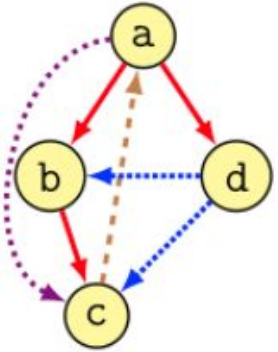


Back edge: c --> a
Does G have a cycle? True

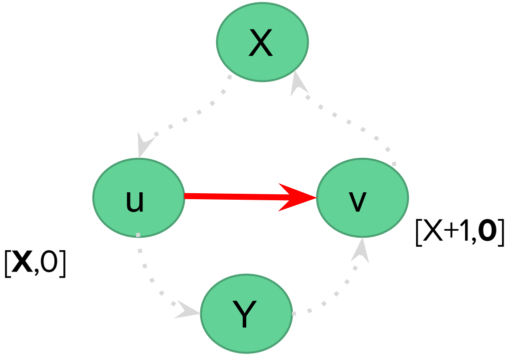


Comment on edge classification

DFS edge
Forward edge
Back edge
Cross edge



Tree edge	$dt[v] == 0$
Back edge:	$dt[u] > dt[v]$ **and** $ft[v] = 0$
Forward edge:	$dt[u] < dt[v]$ **and** $ft[v] \neq 0$
Cross edge:	otherwise



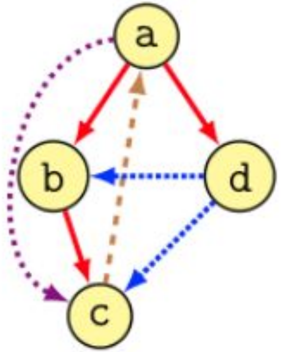
1. if dt[v] == 0, it is the first time we see v in the DFS search. DFS Tree edge!

2. if dt[u] > dt[v] the DFS search found u after v and since the DFS visit started from v is not complete (ft[v] = 0), v is a descendant of u. [Path: v→ X → u]. Back edge!

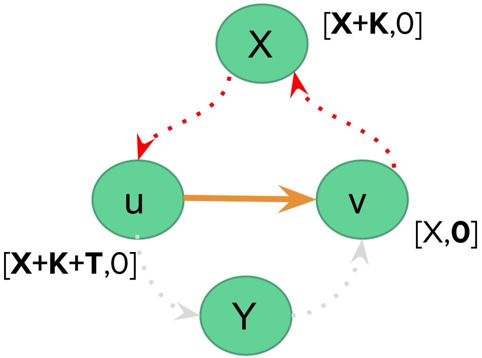
3. if dt[u] < dt[v] the DFS search found v after u, therefore v descends from u. Since the visit of v is complete (ft[v] != 0) this is a Forward edge! [Path: u → Y → v]

Comment on edge classification

DFS edge
Forward edge
Back edge
Cross edge



Tree edge	$dt[v] == 0$
Back edge:	$dt[u] > dt[v]$ **and** $ft[v] = 0$
Forward edge:	$dt[u] < dt[v]$ **and** $ft[v] \neq 0$
Cross edge:	otherwise



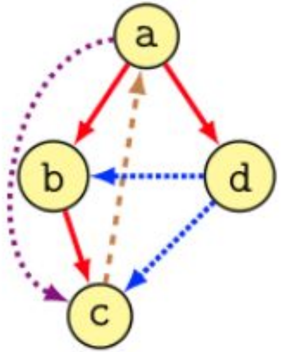
1. if dt[v] == 0, it is the first time we see v in the DFS search. DFS Tree edge!

2. if dt[u] > dt[v] the DFS search found u after v and since the DFS visit started from v is not complete (ft[v] = 0), v is a descendant of u. [Path: v➔ X ➔ u]. Back edge!

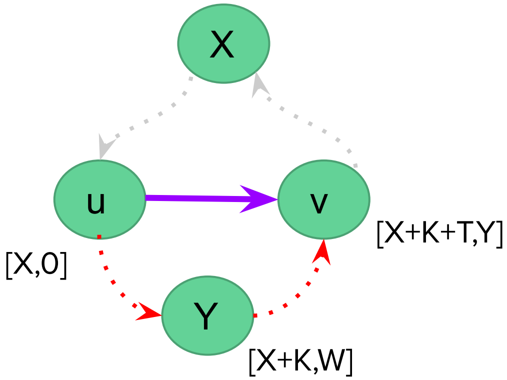
3. if dt[u] < dt[v] the DFS search found v after u, therefore v descends from u. Since the visit of v is complete (ft[v] != 0) this is a Forward edge! [Path: u ➔ Y ➔ v]

Comment on edge classification

DFS edge
Forward edge
Back edge
Cross edge



Tree edge	$dt[v] == 0$
Back edge:	$dt[u] > dt[v]$ **and** $ft[v] = 0$
Forward edge:	$dt[u] < dt[v]$ **and** $ft[v] \neq 0$
Cross edge:	otherwise



1. if dt[v] == 0, it is the first time we see v in the DFS search. DFS Tree edge!

2. if dt[u] > dt[v] the DFS search found u after v and since the DFS visit started from v is not complete (ft[v] = 0), v is a descendant of u. [Path: v➔ X ➔ u]. Back edge!

3. if dt[u] < dt[v] the DFS search found v after u, therefore v descends from u. Since the visit of v is complete (ft[v] != 0) this is a Forward edge! [Path: u ➔ Y ➔ v]

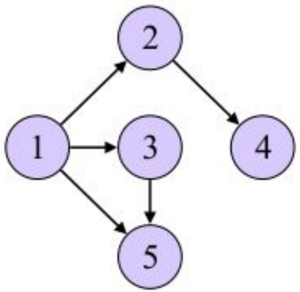
Topological sorting

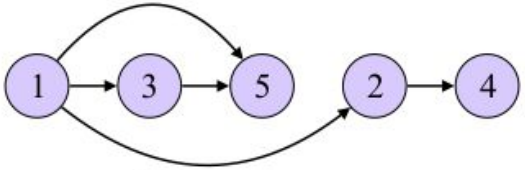
Definition

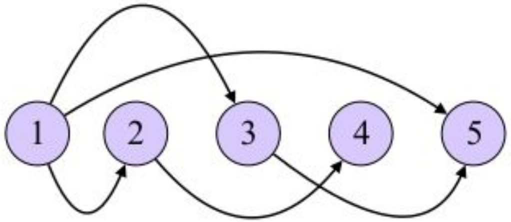
Given a DAG G, a topological sort of G is a linear ordering of its nodes such that if $(u, v) \in E$, then u appears before v in the ordering

Notes:

- There could be several topological sorts
- If there is a cycle, no topological sort is possible







We can think at these DAGs as dependency graphs. If we have edge x-->y activity x has to be completed before y starts.

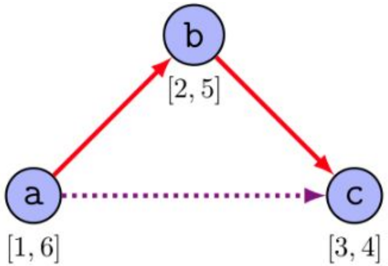
Note: Edges always from left to right: correct order!

Topological sorting

Problem

Write an algorithm that takes a DAG G as input and returns a topological sort of G as output.

How would you solve this problem?



Topological sorting

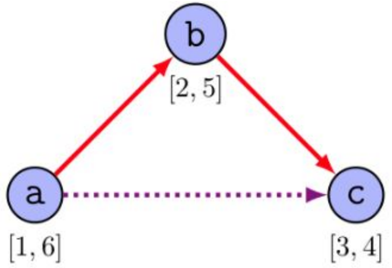
Problem

Write an algorithm that takes a DAG G as input and returns a topological sort of G as output.

How would you solve this problem?

Naive solution

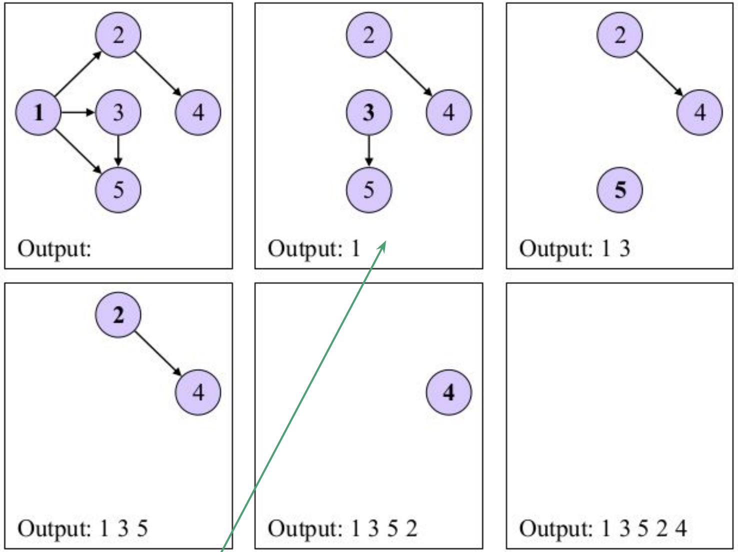
- Find a node u with no incoming edges
- Append u to a list; remove u, together with all its edges
- Repeat the procedure until all nodes have been removed



Topological sorting

Naive solution

- Find a node u with no incoming edges
- Append u to a list; remove u, together with all its edges
- Repeat the procedure until all nodes have been removed



Picking 2 or 3 is equivalent (i.e. originates equivalent topological orderings)

Note: we are destroying the graph!!!
We could make a copy of the graph first, but this is not a great solution...

Topological sorting

Algorithm

- Execute a DFS in which the "visit" operation consists of adding the node at the head of a list "at finish time" (post-order)
- Return the list of nodes obtained in this way

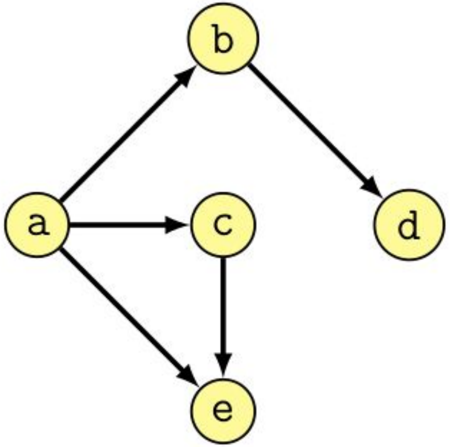
Output

- The sequence of nodes, sorted by decreasing finish time

Why does it work?

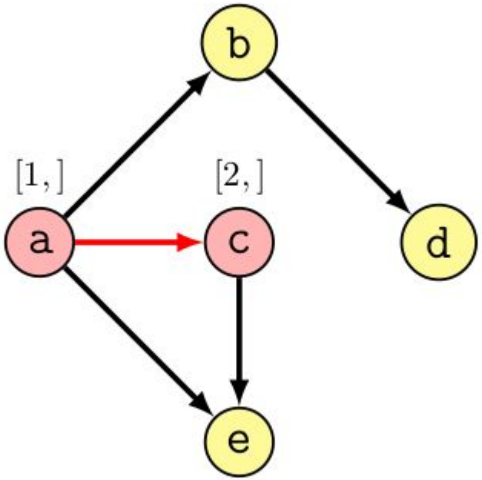
- When a node is "finished", all its descendants have been discovered and added to the list.
- By adding the node in front of the list, nodes are sorted correctly
- We use a stack instead

Topological sorting: example



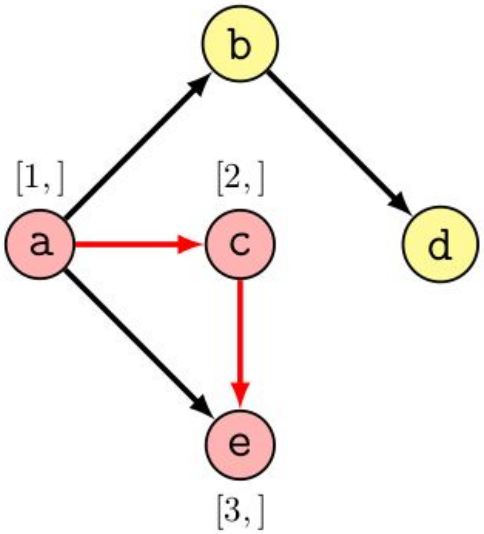
Stack = { }

Topological sorting: example



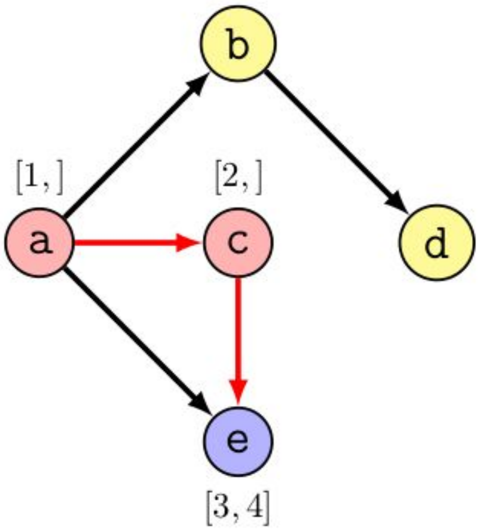
Stack = { }

Topological sorting: example



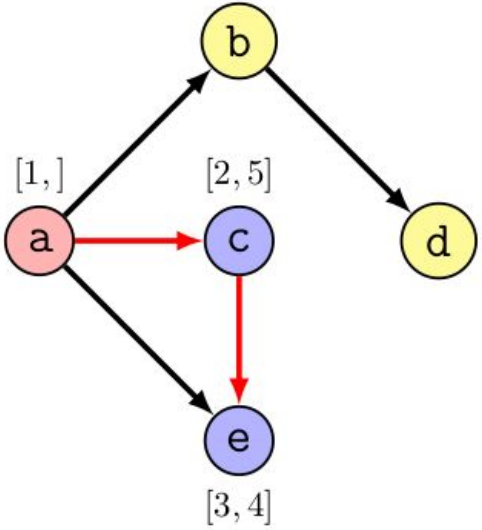
Stack = { }

Topological sorting: example



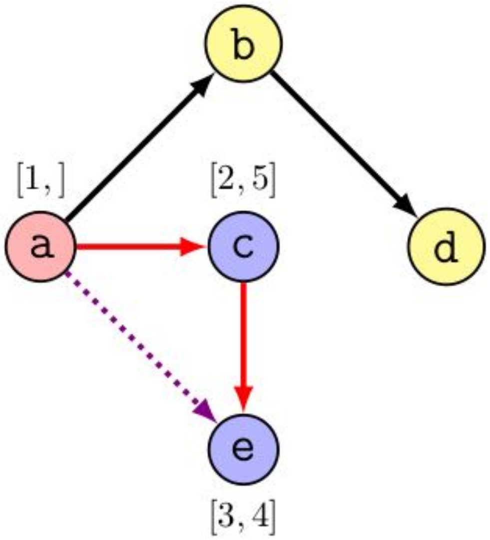
`Stack = { e }`

Topological sorting: example



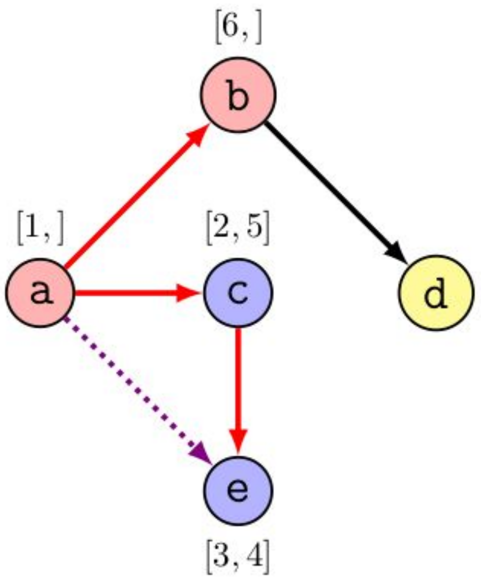
Stack = { c, e }

Topological sorting: example



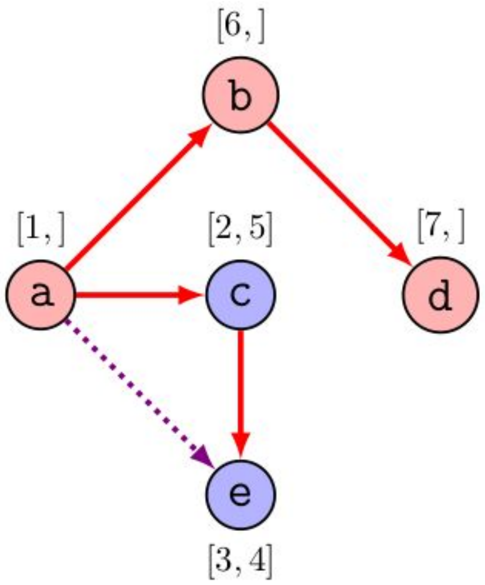
Stack = { c, e }

Topological sorting: example



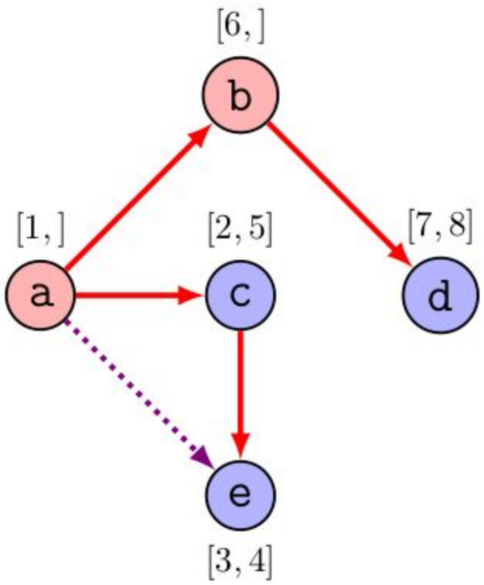
Stack = { c, e }

Topological sorting: example



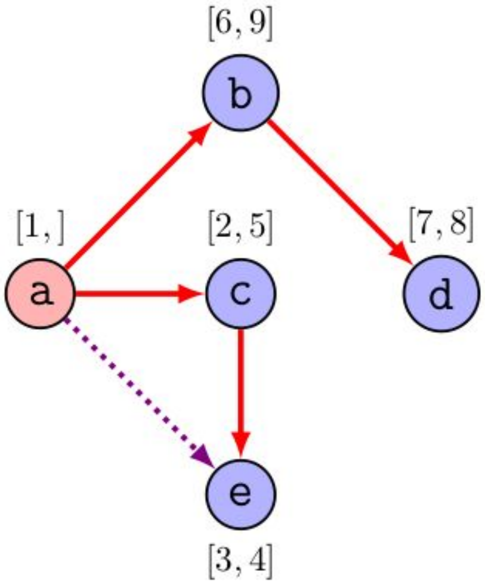
Stack = { c, e }

Topological sorting: example



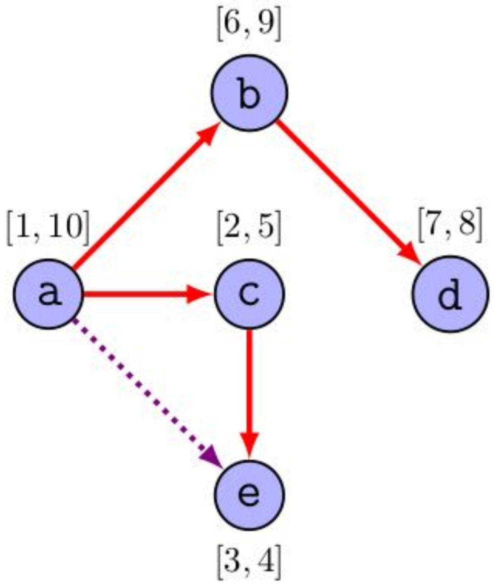
`Stack = { d, c, e }`

Topological sorting: example



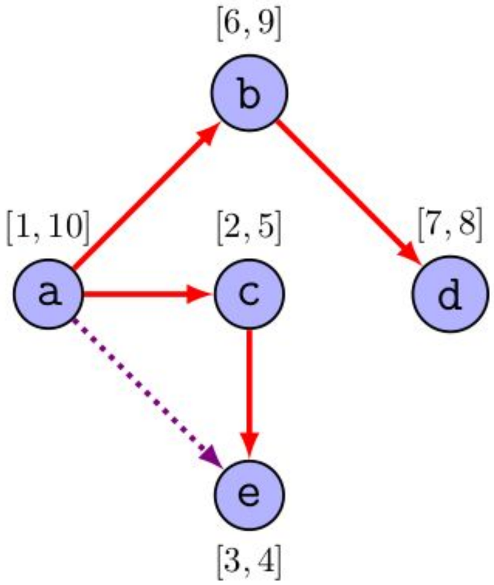
Stack = { b, d, c, e }

Topological sorting: example

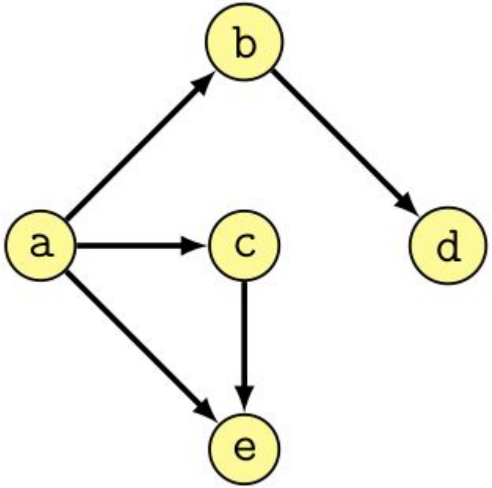


Stack = { a, b, d, c, e }

Topological sorting: example



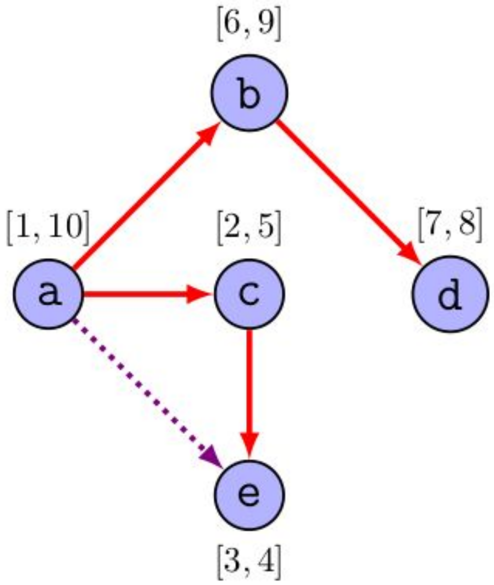
Stack = { a, b, d, c, e }



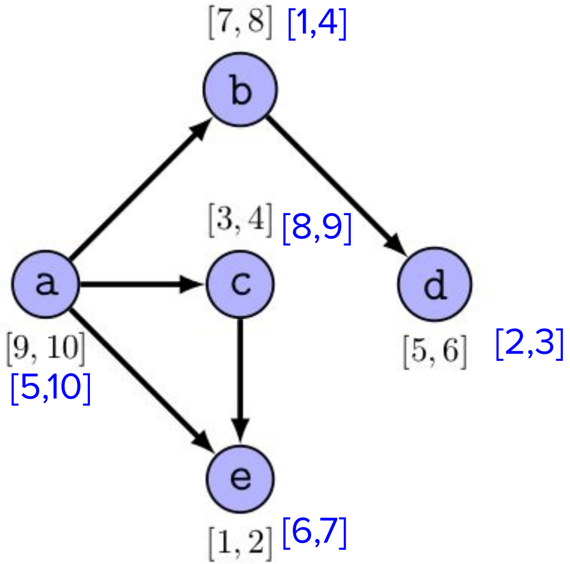
Stack = { }

What happens if nodes are chosen in a different order in the DFS visit?

Topological sorting: example



Stack = { a, b, d, c, e }



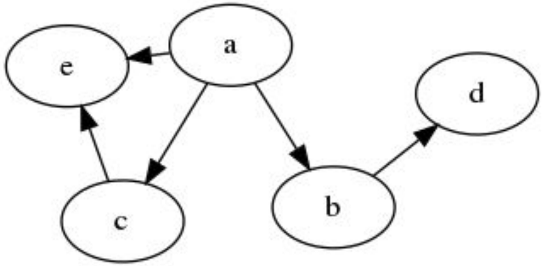
Stack = { a, b, d, c, e }

Stack = {a, c, e, b, d}

What happens if nodes are chosen in a different order in the DFS visit?

Topological sorting: the code

```
def top_sort(G):
    S = Stack()
    visited = set()
    for u in G.node_iterator():
        if u not in visited:
            top_sortRec(G, u, visited, S)
    return S


def top_sortRec(G, u, visited, S):
    visited.add(u)
    for v in G.adj(u):
        if v not in visited:
            top_sortRec(G,v,visited,S)
    S.push(u)
```

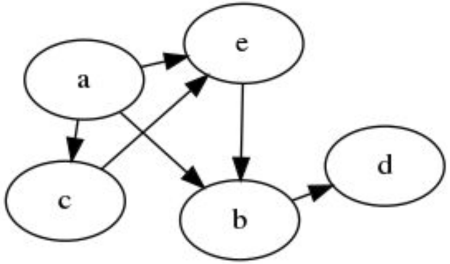



```
G = Graph()
for u,v,c in [('a','c','black'), ('a','b', 'black'), ('c','e','black'), ('a','e', 'black'),
              ('b','d','black')]:

    G.insert_edge(u,v)
print(top_sort(G))
```

```
Stack(a | b | d | c | e)
```

Topological sorting: the code

```
def top_sort(G):
    S = Stack()
    visited = set()
    for u in G.node_iterator():
        if u not in visited:
            top_sortRec(G, u, visited, S)
    return S


def top_sortRec(G, u, visited, S):
    visited.add(u)
    for v in G.adj(u):
        if v not in visited:
            top_sortRec(G,v,visited,S)
    S.push(u)
```




```
G = Graph()
for u,v,c in [('a','b', 'black'), ('a','c','black'), ('a','e', 'black'),
              ('c','e','black'), ('b','d','black'), ('e','b', 'black')]:

    G.insert_edge(u,v)
print(top_sort(G))
```

```
Stack(a | c | e | b | d)
```

Strongly connected graphs and components

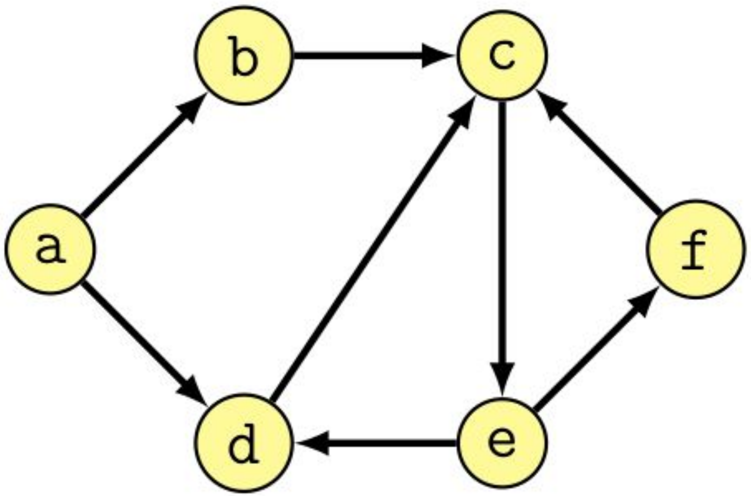
Definitions

- A directed graph $G = (V, E)$ is strongly connected iff every node is reachable from every other node
- A directed graph $G' = (V', E')$ is a strongly connected component iff G' is a connected and maximal subgraph of G
- G' is a subgraph of G ($G' \subseteq G$) iff $V' \subseteq V$ and $E' \subseteq E$
- G' is maximal iff there is not other graph G'' of G such that G'' is strongly connected and larger than G' (i.e. $G' \subseteq G'' \subseteq G$)

Strongly connected graphs and components

Question

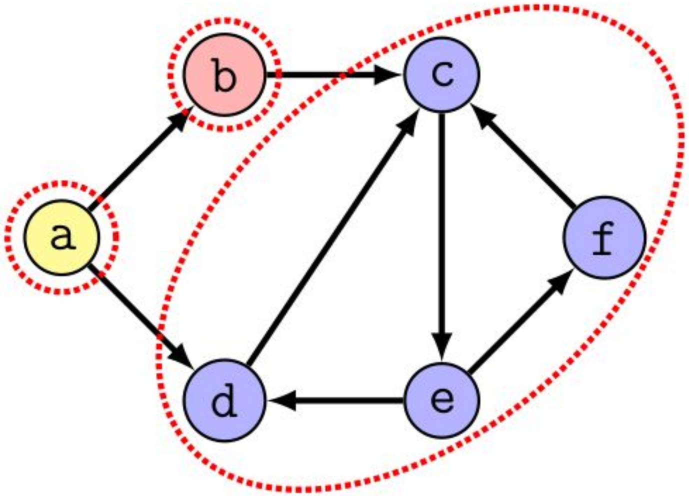
- What are the strongly connected components of this graph?



Strongly connected graphs and components

Question

- What are the strongly connected components of this graph?

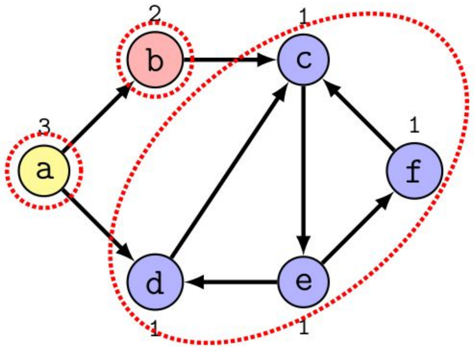


Naive (and wrong!) solution

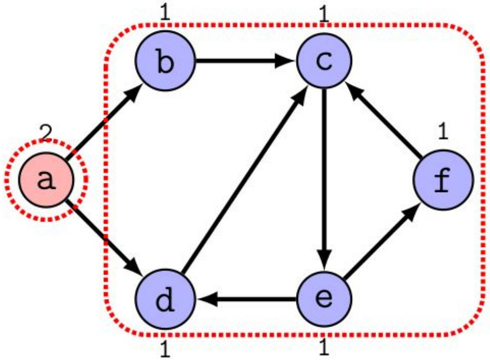
- Just apply the CC algorithm to directed graphs
- The result depends on the starting node

```
def cc(G):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    for u in G.node_iterator():
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```

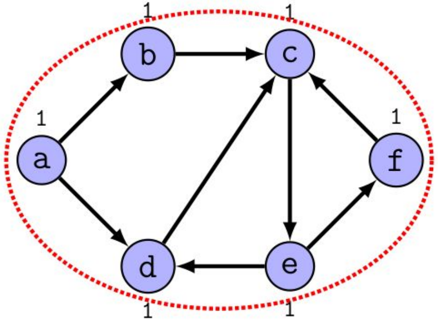
In a nutshell: perform a DSF visit, assign to each visit the same component number until all nodes visited



DFS visit starting from C, then from B, then from A



DFS visit starting from B, then from A



DFS visit starting from A

Strongly connected components algorithm

Kosaraju Algorithm (1978)

- Perform a DFS of G
- Compute the transpose graph G_T
- Run the connected component algorithm on G_T, examining the nodes in decreasing finish time w.r.t. the first visit
- Returns the identifiers of the nodes

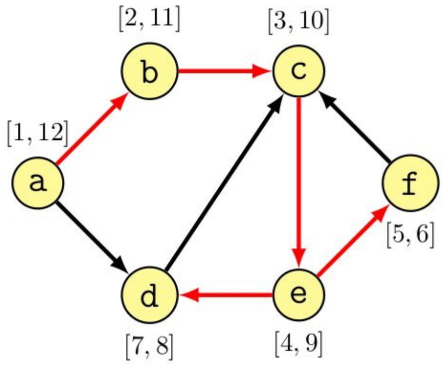
```
def scc(G):
    #performs a topological sort of G
    S = top_sort(G)
    #Transposes G
    GT = transpose(G)
    #modified version of CC algo that
    #gets starting nodes off the stack S
    counter, ids = cc(GT,S)
```

Topological sorting of general graphs

By applying the topological sort algorithm on a general graph, we are sure that:

- if an edge (u, v) does not belong to a cycle, than u appears before v in the sorted sequence

We use thus `topsort()` to obtain nodes in decreasing finish time.



`Stack = { a, b, c, e, d, f }`

NOTE: we might have cycles, so this does not necessarily mean that we obtain a topological sort!!!

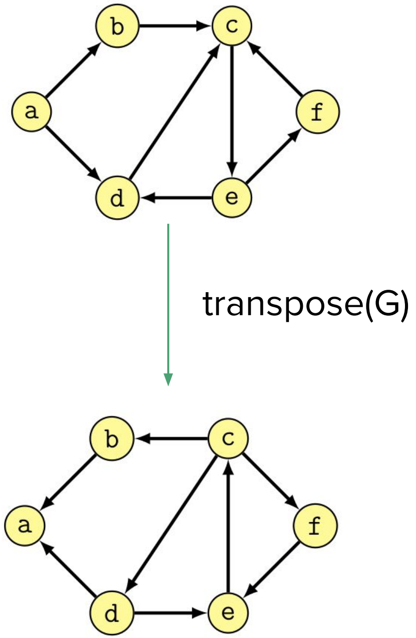
But the important thing is that <u>all the nodes before the cycle(s) and after the cycles(s) are put in the correct topological sort</u>.

Transpose of a graph

Given a graph $G = (V, E)$, the transpose graph $G_T = (V, E_T)$ has the same nodes, while edges are directed in the opposite way:

$$E_T = \{(u, v) \mid (v, u) \in E\}$$

```python
def transpose(G):
    tmpG = Graph()
    for u in G.node_iterator():
        for v in G.adj(u):
            tmpG.insert_edge(v,u)
    return tmpG
```



Transpose of a graph

Given a graph $G = (V, E)$, the transpose graph $G_T = (V, E_T)$ has the same nodes, while edges are directed in the opposite way:

$$E_T = \{(u, v) \mid (v, u) \in E\}$$

```
def transpose(G):
    tmpG = Graph()
    for u in G.node_iterator():
        for v in G.adj(u):
            tmpG.insert_edge(v,u)
    return tmpG
```

$\boxed{\text{Computational cost: } O(m+n)}$

- $O(n)$ nodes added
- $O(m)$ edges added
- Each add operation costs $O(1)$

Modified connected components

Instead of examining the nodes in an arbitrary order, this version of cc(G,S) examines them in the order in which they are stored in the stack S.

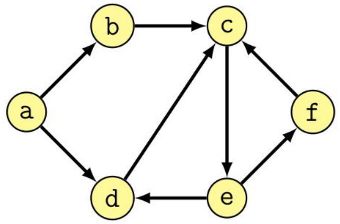
```
def cc(G, S):
    ids = dict()
    for node in G.node_iterator():
        ids[node] = 0
    counter = 0
    while len(S) > 0:
        u = S.pop()
        if ids[u] == 0:
            counter += 1
            ccdfs(G, counter, u, ids)
    return (counter, ids)

def ccdfs(G, counter, u, ids):
    ids[u] = counter
    for v in G.adj(u):
        if ids[v] == 0:
            ccdfs(G, counter, v, ids)
```

Computational cost: $O(m + n)$

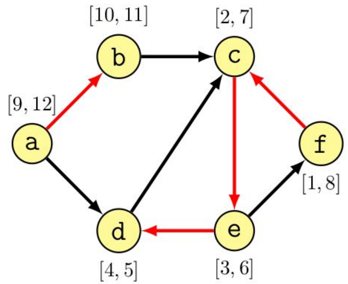
Each phase requires $O(m + n)$

Putting it all together

```
def scc(G):
    #performs a topological sort of G
    S = top_sort(G)
    #Transposes G
    GT = transpose(G)
    #modified version of CC algo that
    #gets starting nodes off the stack S
    counter, ids = cc(GT,S)
```

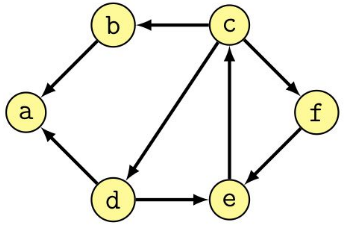


top_sort(G)

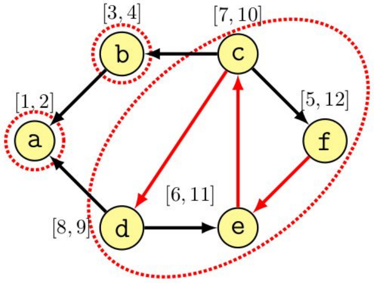


Stack = { a, b, f, c, e, d }

transpose(G)



cc(GT,S)



Stack = { a, b, f, c, e, d }

Output:
Components: 3
Ids:{'b': 2, 'a': 1, 'd': 3, 'c': 3, 'e': 3, 'f': 3}

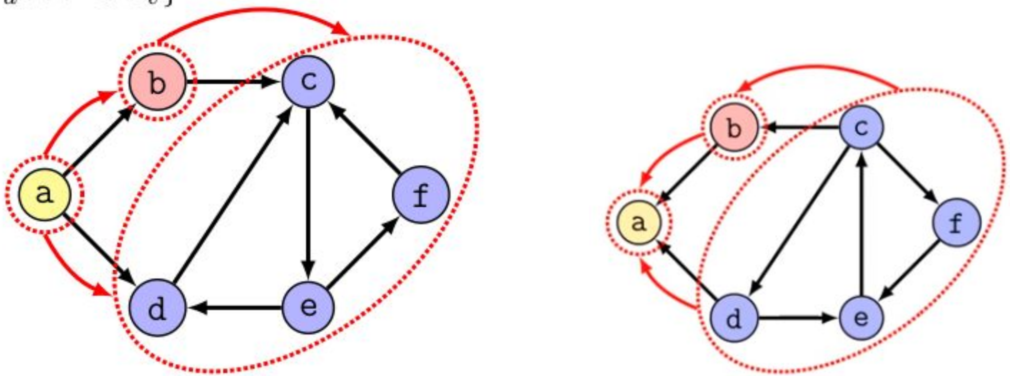
Proof of correctness...

Component Graph $V_c = (V_c, E_c)$

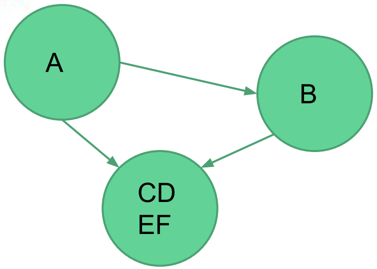
- $V_c = \{C_1, C_2, \ldots, C_k\}$, where C_i is the i-th SCC of G
- $E_c = \{(C_u, C_v) | \exists (u, v) \in E \wedge u \in C_u \wedge v \in C_v\}$

Questions

- What is the relationship between the SCCs of G and the SCCs of G_T?
- Is the component graph acyclic?



YES. Otherwise any cycle would be a bigger SCC.



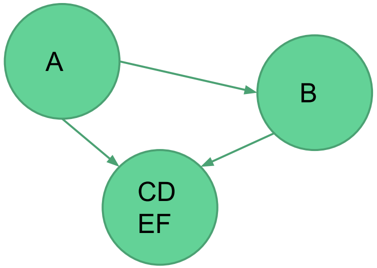
NO CYCLES: top_sort correctly sorts the components

Proof of correctness...

Discovery time and finish for the component graph

$$dt(C) = \min\{dt(u) | u \in C\}$$
$$ft(C) = \max\{ft(u) | u \in C\}$$

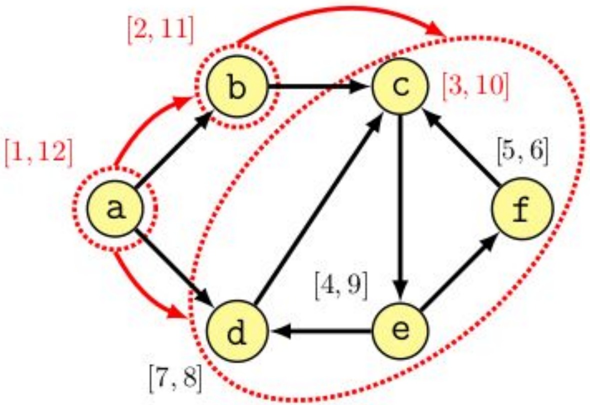
These discovery/finish times correspond to the discovery/finish time of the first node to be visited in component C

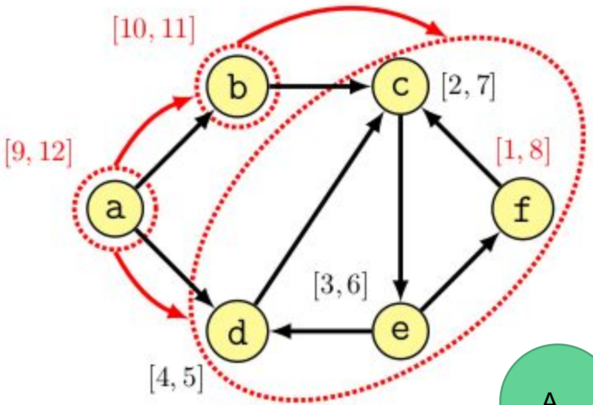


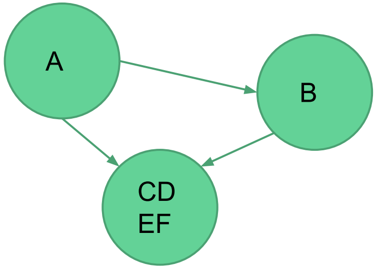
Proof of correctness...

Theorem

Let C and C' be two distinct SCCs in the directed graph $G = (V, E)$. If there is an edge $(C, C') \in E_c$, then $ft(C) > ft(C')$.





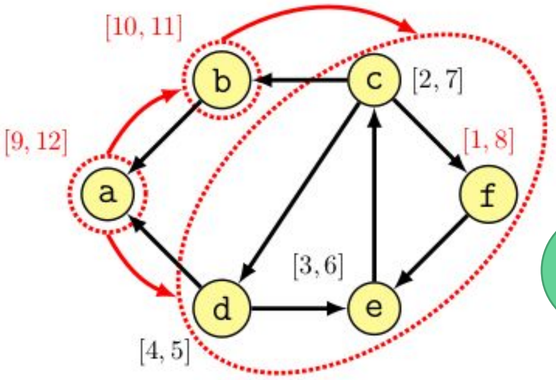


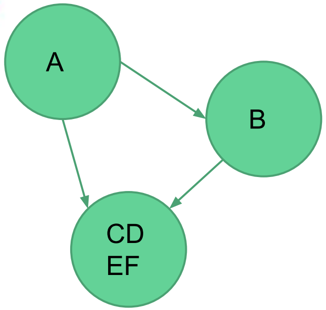
Proof of correctness...

Corollary

Let C_u and C_v be two distinct SCCs in the directed graph $G = (V, E)$.
If there is an edge $(u, v) \in E_t$ with $u \in C_u$ and $v \in C_v$, then $ft(C_u) < ft(C_v)$.

$$(u, v) \in E_t \Rightarrow$$
$$(v, u) \in E \Rightarrow$$
$$(C_v, c_u) \in E_c \Rightarrow$$
$$ft(C_v) > ft(C_u) \Rightarrow$$
$$ft(C_u) < ft(C_v)$$



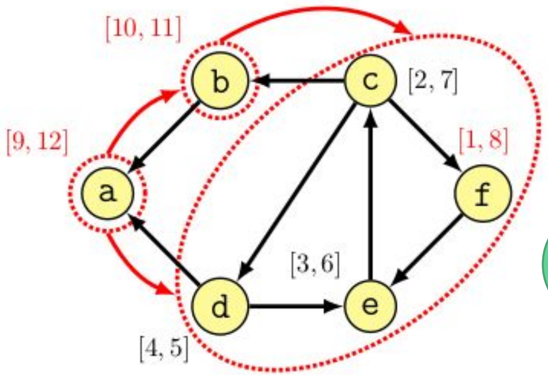


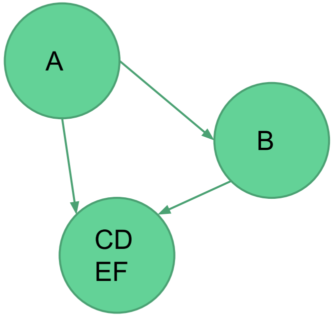
Proof of correctness...

Corollary

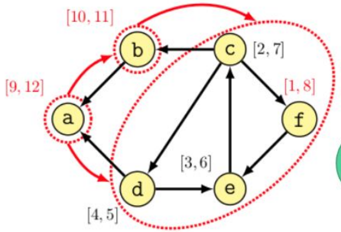
Let C_u and C_v be two distinct SCCs in the directed graph $G = (V, E)$.
If there is an edge $(u, v) \in E_t$ with $u \in C_u$ and $v \in C_v$, then $ft(C_u) < ft(C_v)$.

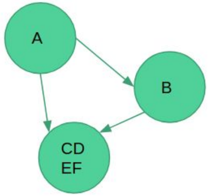
$$
\begin{aligned}
(b, a) \in E_t &\Rightarrow \\
(a, b) \in E &\Rightarrow \\
(C_a, C_b) \in E_c &\Rightarrow \\
12 = ft(C_a) > ft(C_b) = 11 &\Rightarrow \\
11 = ft(C_b) < ft(C_a) = 12 &
\end{aligned}
$$





Proof of correctness…





- If the component C_u and the component C_v are connected by an edge $(u, v) \in E_t$, then:
  - From the corollary, $ft(C_u) < ft(C_v)$
  - From the algorithm, the visit of C_v will start before the visit of C_u
- There is no path between C_v and C_u in G_t (otherwise the graph would be cyclic)
  - From the algorithm, the visit of C_v will not reach C_u,

In other words, `cc()` will correctly assign the component identifiers to nodes.

If you are starting to have fun...

Good news... there are at least 110+ other algorithms on graphs!

Pages in category "Graph algorithms"

The following 118 pages are in this category, out of 118 total. This list may not reflect recent changes (learn more).

A

- A* search algorithm
- Alpha–beta pruning
- Aperiodic graph

B

- B*
- Barabási–Albert model
- Belief propagation
- Bellman–Ford algorithm
- Bianconi–Barabási model
- Bidirectional search
- Borůvka's algorithm
- Bottleneck traveling salesman problem
- Breadth-first search
- Bron–Kerbosch algorithm
- Bully algorithm

C

- Centrality
- Chaitin's algorithm
- Christofides algorithm
- Clique percolation method
- Closure problem
- Color-coding
- Contraction hierarchies
- Courcelle's theorem

- Floyd–Warshall algorithm
- Force-directed graph drawing
- Ford–Fulkerson algorithm
- Fringe search

G

- Gallai–Edmonds decomposition
- Girvan–Newman algorithm
- Goal node (computer science)
- Gomory–Hu tree
- Graph bandwidth
- Graph edit distance
- Graph embedding
- Graph isomorphism
- Graph isomorphism problem
- Graph kernel
- Graph reduction
- Graph traversal

H

- Havel–Hakimi algorithm
- Hierarchical closeness
- Hierarchical clustering of networks
- Hopcroft–Karp algorithm

I

- Iterative deepening A*
- Initial attractiveness

- Minimum bottleneck spanning tree
- Misra & Gries edge coloring algorithm

N

- Nearest neighbour algorithm
- Network flow problem
- Network simplex algorithm
- Nonblocking minimal spanning switch

P

- PageRank
- Parallel all-pairs shortest path algorithm
- Parallel breadth-first search
- Path-based strong component algorithm
- Pre-topological order
- Prim's algorithm
- Proof-number search
- Push–relabel maximum flow algorithm

R

- Reverse-delete algorithm
- Rocha–Thatte cycle detection algorithm

S

- Semantic Brand Score
- Sethi–Ullman algorithm
- Shortest Path Faster Algorithm
- SMA*