Jan 2021 | © Guven

Finding Articulation Points with DFS

This notebook demonstrates,

1. Determining the articulation points using the algorithm provided in class
2. An example with 14 nodes to be traversed by DFS
3. The edges are provided as the input with 2-tuples of 2 labels which correspond to node names
4. Example code will convert the input to an adjacency list, i.e. a dictionary with keys as vertex labels and values as the **Vertex class** which has **label**, **dfs** and **parent** fields
5. DFS will start from a given root and traverse the graph.
6. Uses `networkx` library to draw the graphs

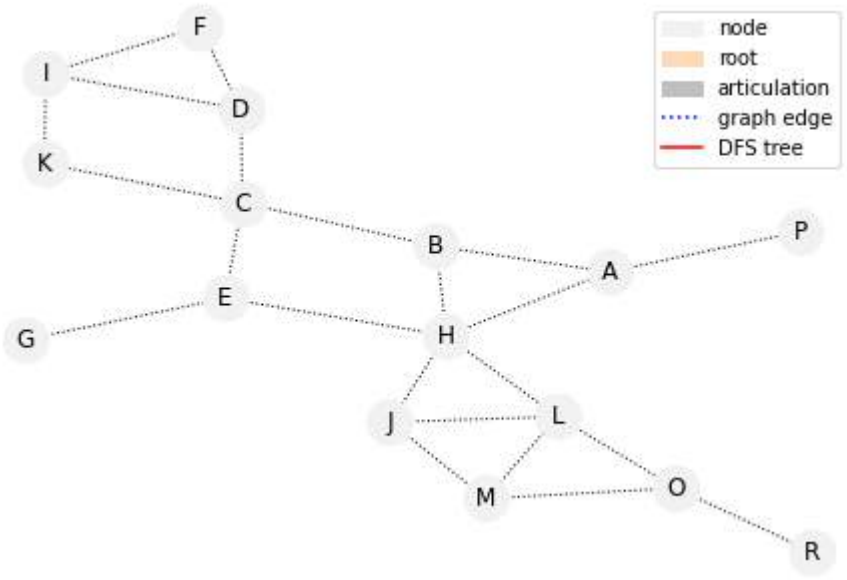
(ref: Tarjan, Robert. "Depth-first search and linear graph algorithms." SIAM journal on computing 1.2 (1972): 146-160.)

In [1]:
```
%matplotlib inline
import warnings
import matplotlib.cbook
import matplotlib.pyplot as plt
import networkx as nx
print(f'networkx version {nx.__version__}')
```

```
networkx version 2.4
```

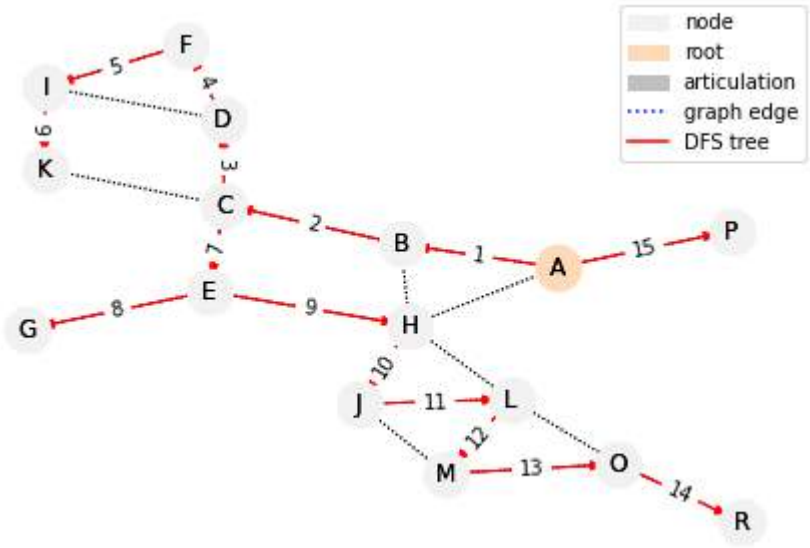
In [2]:
```
Edges = [('A','B'),('A','H'),('B','C'),('B','H'),('C','D'),('C','E'),
('D','F'),('D','I'),('C','K'),
         ('E','G'),('E','H'),('F','I'),('H','J'),('H','L'),('I','K'),
('J','L'),('L','M'),
         ('L','O'),('J','M'),('M','O'),('A','P'),('R','O')]

# Set a vertex class to store DFS (first timestamp) and parent
class Vertex:
    def __init__(self, label):
        self.label=label; self.dfs=None; self.p=None; self.low=None;
self.seen=False

# Root is specially handled
ROOT_PARENT_LABEL = 'nul'
ROOT_PARENT = Vertex(ROOT_PARENT_LABEL)

# node layout as a function
G_Layout_func = nx.kamada_kawai_layout

def adj_list(_e):  # _e is a list of 2-tuples
    # Generate an adjacency list from the input Edges
    from collections import defaultdict
    #
    edges = defaultdict(list)
    vertices = {}  # convert the labels to vertex objects
    for v1, v2 in _e:
        if v1 not in vertices:
            vertices[v1] = Vertex(v1)
        if v2 not in vertices:
            vertices[v2] = Vertex(v2)
        edges[v1] += [vertices[v2]]
        edges[v2] += [vertices[v1]]
    return vertices, edges

def draw_tree(_vertices, _edges, _arts=None):
    import matplotlib.patches as mpatches
    import matplotlib.lines as mlines
    #
    root = []
    # g will show entire graph, back-edges dotted
    g = nx.Graph()
    g.add_edges_from(_edges)
    pos = G_Layout_func(g)  # node positions, shared among all graphs
    # tree edges
    e2, edgelabel = [], {}
    for v in _vertices.values():
        if v.p is not None:
            if v.p.label == ROOT_PARENT_LABEL:  # handle root as spec
ial
                root = [v.label]
                continue
            e2 += [(v.p.label, v.label)]
            edgelabel[(v.p.label, v.label)] = str(v.dfs-1)
    g_di = nx.DiGraph()
    g_di.add_edges_from(e2)
```

```
    nx.draw(g, pos, node_size=500, node_color='0.95', with_labels=Tru
e, style=':')
    if _arts is not None:
        nx.draw_networkx_nodes(g, pos, node_size=500, node_color='0.7
5', nodelist=_arts)
    #
    nx.draw(g, pos, node_size=500, node_color='peachpuff', nodelist=r
oot, edgelist=[])
    nx.draw_networkx_edges(g_di, pos, edge_color='r')
    nx.draw_networkx_edge_labels(g_di, pos, edgelabel)
    #
    p1 = mpatches.Patch(color='0.95', label='node')
    p2 = mpatches.Patch(color='peachpuff', label='root')
    p3 = mpatches.Patch(color='0.75', label='articulation')
    p4 = mlines.Line2D([], [], color='b', ls=':', label='graph edge')
    p5 = mlines.Line2D([], [], color='r', label='DFS tree')
    plt.legend(handles=[p1, p2, p3, p4, p5])

plt.show()
#
vertices, edges = adj_list(Edges)
draw_tree(vertices, Edges)
```



Following use 'A' as the root node.

In [3]:
```
# DFS uses a stack iteratively to avoid recursion, pre-order traversa
l
# edge list key is the vertex label
def dfs(_e, _root):
    # pre-order traversal to populate dfs values
    dfscounter = 1
    stack = [_root]  # stack is simply a Python list
    while len(stack) > 0:
        v1 = stack.pop()
        if not v1.seen:  # not visited yet
            v1.seen = True
            v1.dfs = dfscounter
            dfscounter += 1
            # edge dictionary key is vertex label, value is list of n
odes
            for v2 in sorted(_e[v1.label], key=lambda _x: _x.label, r
everse=True):
                if not v2.seen:  # not visited yet
                    v2.p = v1
                    stack += [v2]  # set parent

# DFS expects vertex objects
vertices1, edges1 = adj_list(Edges)

# Set the root
root = vertices1['A']
root.p = ROOT_PARENT

dfs(edges1, root)

draw_tree(vertices1, Edges)
draw_tree(vertices1, Edges)
```

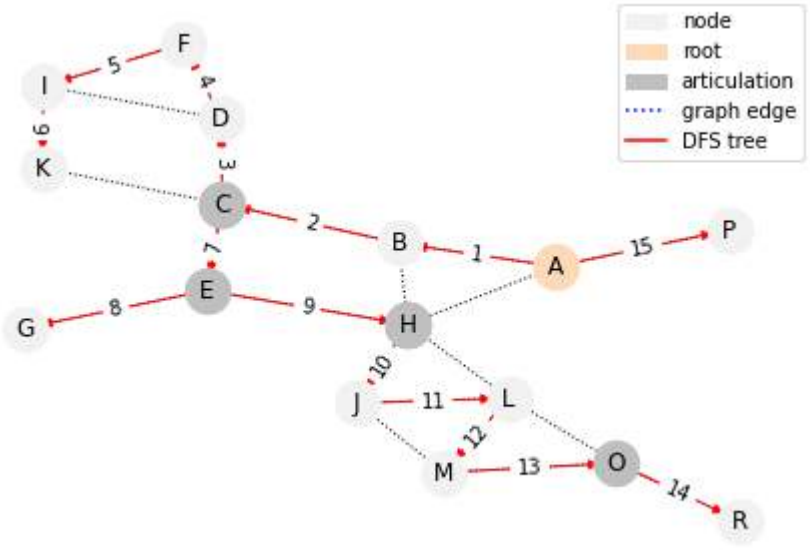


In [4]:

```
# Find low values on the DFS tree
def dfs2(_e, _root):
    # pre-order traversal to populate DFS values
    dfscounter = 1
    stack = [_root]  # stack is simply a Python list
    while len(stack) > 0:
        v1 = stack.pop()
        if not v1.seen:  # not visited yet
            v1.seen = True
            v1.dfs = dfscounter
            dfscounter += 1
            # edge dictionary key is vertex label, value is list of n
odes
            for v2 in (_v for _v in sorted(_e[v1.label],
                    key=lambda _x: _x.label, reverse=True) if not _v.
seen):
                v2.p = v1
                stack += [v2]  # set parent
    print('pre-order traversal completed')

    # reset visited field to DFS traverse the graph again
    for _v in set([_v for _o in _e.values() for _v in _o]):
        _v.seen = False

    # post-order traversal ON THE TREE to populate low values
    _root.low = _root.dfs  # there is no ancestor for root
    stack = [_root]
    while len(stack) > 0:
        v1 = stack[-1]
        vs = [_v for _v in _e[v1.label] if not _v.seen and _v.p == v1
]
        if len(vs) > 0:
            for v2 in vs:
                v2.seen = True
                v2.low = v2.dfs
                stack += [v2]
        else:
            v1 = stack.pop()
            if v1 == _root:  # handle root as special
                continue

            # check for back-edges
            for v2 in (_v for _v in _e[v1.label] if v1.p != _v):
                v1.low = min(v1.low, v2.dfs)

            # check for child low value and update parent
            v1.p.low = min(v1.p.low, v1.low)

    print('post-order traversal completed')

    # check children low values to see if the vertex is an articulati
on point
    art_points = []
    for vlist in _e.values():
        for v in vlist:
            if v != _root and v.p != _root and v.low >= v.p.dfs:
```

```
                art_points += [v.p.label]

    return art_points


# print the dfs values and low values
def dfslow_info(_vertices, _Edges, _art_points):
    draw_tree(_vertices, _Edges, _art_points)
    for v in sorted(_vertices.values(), key=lambda _x: _x.label):
        print(v.label, v.dfs, v.low)
    print(f'articulation points= {set(_art_points)}')

# DFS expects vertex objects
vertices1, edges1 = adj_list(Edges)

# Set the root
root = vertices1['A']
root.p = ROOT_PARENT

# New DFS
art_points = dfs2(edges1, root)

dfslow_info(vertices1, Edges, art_points)
```

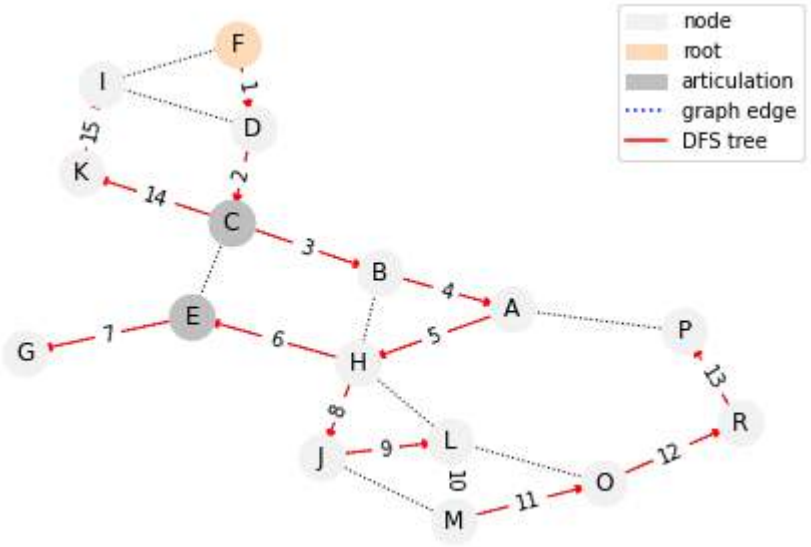
```
pre-order traversal completed
post-order traversal completed
A 1 1
B 2 1
C 3 1
D 4 3
E 8 1
F 5 3
G 9 9
H 10 1
I 6 3
J 11 10
K 7 3
L 12 10
M 13 11
O 14 12
P 16 16
R 15 15
articulation points= {'O', 'C', 'E', 'H'}
```



Let's add an edge between 'P' and 'R'.

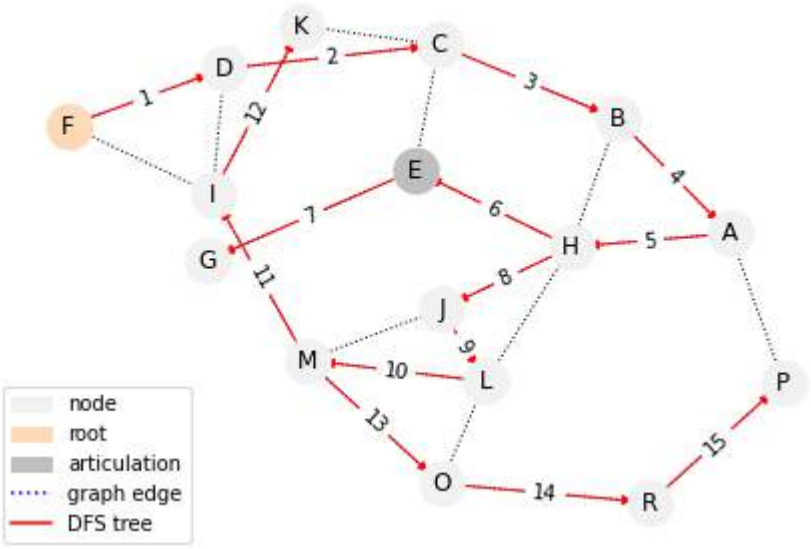
In [5]:
```
Edges2 = Edges + [('P','R')]
vertices2, edges2 = adj_list(Edges2)

# Set the root
root = vertices2['F']
root.p = ROOT_PARENT

art_points2 = dfs2(edges2, root)

dfslow_info(vertices2, Edges2, art_points2)
```

```
pre-order traversal completed
post-order traversal completed
A 5 3
B 4 3
C 3 1
D 2 1
E 7 3
F 1 1
G 8 8
H 6 3
I 16 1
J 9 5
K 15 1
L 10 5
M 11 5
O 12 5
P 14 5
R 13 5
articulation points= {'C', 'E'}
```



In [6]:
```
Edges3 = Edges2 + [('I','M')]
vertices3, edges3 = adj_list(Edges3)

root = vertices3['F']
root.p = ROOT_PARENT

art_points3 = dfs2(edges3, root)

dfslow_info(vertices3, Edges3, art_points3)
```

```
pre-order traversal completed
post-order traversal completed
A 5 1
B 4 1
C 3 1
D 2 1
E 7 3
F 1 1
G 8 8
H 6 1
I 12 1
J 9 1
K 13 3
L 10 1
M 11 1
O 14 5
P 16 5
R 15 5
articulation points= {'E'}
```



Exercises

- Check why 'E' is an articulation point
- Add code to include marking the root if it is an articulation point
- Add code to find bridges